AD-A054 377 COMMAND AND CONTROL TECHNICAL CENTER WASHINGTON D C F/G 15/7
THE CCTC QUICK-REACTING GENERAL WAR GAMING SYSTEM. (QUICK). PRO--ETC(U)
JUN 77 D J SANDERS, P F MAYKRANTZ, J M HERRIN
UNCLASSIFIED CCTC-CSM-MM-9-77-V1-PT-1 SBIE-AD-E100 051 NL

1 OF 5
AD
A054377

AD-E100 051

COMPUTER SYSTEM MANUAL
CSM MM 9-77
VOLUME I, PART I
1 JUNE 1977

13

FOR FURTHER TRAN

Part II
A054 310

AD A054377

# CCTC

Vol II
AD50 081

COMMAND
& CONTROL
TECHNICAL
CENTER

DDC
RECEIVED
MAY 31 1978
B

# THE CCTC QUICK-REACTING WAR GAMING SYSTEM (QUICK)

## PROGRAM MAINTENANCE MANUAL

## VOLUME I - DATA MANAGEMENT SUBSYSTEM

AD No.
DDC FILE COPY

DEFENSE
COMMUNICATIONS
AGENCY

THIS DOCUMENT HAS BEEN APPROVED FOR PUBLIC RELEASE AND SALE; ITS DISTRIBUTION IS UNLIMITED.

## RECORD OF CHANGES

| CHANGE NUMBER | DATED | DATE ENTERED | SIGNATURE OF PERSON MAKING CHANGE |
|---|---|---|---|
| | | | CCTC should be corporate author per telecon w/Proj. officer, CCTC, C. G. Thompson, 16 Jun 78. |

MB Bonnett, DCA was notified per ODL Instructions

COMMAND AND CONTROL TECHNICAL CENTER

14 CCTC-CSM-MM-9-77-V1-PT-1

18 SBIE

9 Computer System Manual, CSM-MM-9-77 19 AD-E100 051

11 1 June 1977 12 448p.

6

# THE CCTC QUICK-REACTING GENERAL WAR GAMING SYSTEM (QUICK),

## Program Maintenance Manual,

## Volume I - Data Management Subsystem,

## Part I,

10 Dale J. Sanders, Paul F. M. Maykrantz, Jim M. Herrin Edward F. Bersson

OK

SUBMITTED BY:

C. G. Thompson

C. G. THOMPSON
Project Officer

APPROVED BY:

R E Harshbarger

R. E. HARSHBARGER
Acting Deputy Director
NMCS ADP

15 DCA100-75-C-0019

Copies of this document may be obtained from the Defense Documentation Center, Cameron Station, Alexandria, Virginia 22314.

Approved for public release; distribution unlimited.

409658

JOB

## ACKNOWLEDGMENT

This documentation was prepared under the direction of the Chief for Military Studies and Analysis, CCTC, in response to a requirement of the Studies, Analysis, and Gaming Agency, Organization of the Joint Chiefs of Staff. Technical support was provided by System Sciences, Incorporated under Contract Number DCA100-75-C-0019.

ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

CONTENTS

Part I

Section Page

| Section | | Page |
|---|---|---|

Section Page

ILLUSTRATIONS (PART I)

| Figure | | Page |
|---|---|---|
| 47 | Subroutine CHANGE: Step Five | 291 |
| 48 | Subroutine CHANGE: Step Six | 292 |
| 49 | Subroutine CHANGE: Step Seven | 296 |
| 50 | Subroutine CHANGE: Step Eight | 302 |
| 51 | Subroutine DESSCH | 311 |
| 52 | Subroutine NXTDES | 317 |
| 53 | Subroutine CREAAT: Step 1 | 319 |
| 54 | Subroutine CREAAT: Step 2 | 322 |
| 55 | Subroutine CREAAT: Step 3 | 329 |
| 56 | Subroutine CREAAT: Step 4 | 331 |
| 57 | Subroutine CREAAT: Step 5 | 336 |
| 58 | Subroutine CREAAT: Step 6 | 343 |
| 59 | Subroutine CREAAT: Step 7 | 345 |
| 60 | Subroutine CREAAT: Step 8 | 349 |
| 61 | Subroutine CREAAT: Step 9 | 351 |
| 62 | Subroutine CREAAT: Step 10 | 352 |
| 63 | Subroutine CREAAT: Step 11 | 353 |
| 64 | Subroutine CREAAT: Step 12 | 357 |
| 65 | Subroutine CREAAT: Step 13 | 358 |
| 66 | Subroutine CREAAT: Step 14 | 360 |
| 67 | Subroutine CREAAT: Step 15 | 363 |
| 68 | Subroutine DELETE | 371 |
| 69 | Subroutine ENTMOD (EDIT) | 388 |
| 70 | Subroutine COUNTS | 392 |
| 71 | Subroutine GENEDIT | 400 |
| 72 | Subroutine BUILDTAB | 403 |
| 73 | Subroutine FORMLOC | 406 |
| 74 | Subroutine SETFLD | 408 |
| 75 | Subroutine SWITH | 413 |
| 76 | Subroutine NORMAL | 418 |
| 77 | Subroutine PROCEDIT | 421 |
| 78 | Subroutine XWITH | 425 |

TABLES (PART I)

# ABSTRACT

The computerized Quick-Reacting General War Gaming System (QUICK) will accept input data, automatically generate global strategic nuclear war plans, provide output summaries, and produce tapes to simulator subsystems external to QUICK. QUICK has been programmed in FORTRAN for use on the CCTC HIS 6000 computer system.

The QUICK Maintenance Manual consists of four volumes: Volume I, Data Management Subsystem; Volume II, Weapon/Target Identification Subsystem; Volume III, Weapon Allocation Subsystem, Volume IV, Sortie Generation Subsystem. The Maintenance Manual complements the other QUICK Computer System Manuals to facilitate application of the war gaming system. This volume, Volume I, in two parts, provides the programmer/analyst with a technical description of the purpose, functions, general procedures, and programming techniques applicable to the modules (programs) and subroutines of the Data Management subsystem. Companion documents are:

a. USERS MANUAL
   Computer System Manual CSM UM 9-77, Volume I
   Computer System Manual CSM UM 9-77, Volume II
   Computer System Manual CSM UM 9-74, Volume III
   Computer System Manual CSM UM 9-74, Volume IV
   Provides detailed instructions for applications of the system

b. TECHNICAL MEMORANDUM
   Technical Memorandum TM 153-77
   Provides a nontechnical description of the system for senior management personnel

## SECTION 1. GENERAL

### 1.1 Purpose

This volume of the QUICK Program Maintenance Manual describes the modules which are part of the QUICK Data Management subsystem, and provides a description of the data base, executive software, subroutines, and functions which comprise the QUICK utility package. The information contained herein is presented on a module-by-module basis, complemented with discussions on a computer programming maintenance subject-by-subject basis. Initially, however, the data base is outlined. The module-by-module discussions are structured so that a maintenance programmer can understand the module functions and programming techniques. The computer subjects are structured to inform the maintenance programmer of overall system programming techniques and conventions.

Subsequent subsections present general descriptions of the overall QUICK system and Data Management subsystem, the equipment, and module environment, and the organization of this volume.

### 1.2 Program Environment - QUICK System Overview

The QUICK-Reacting General War Gaming System (QUICK) is a unique analytical tool which provides a comprehensiveness to strategic war gaming that has not been available through other computerized models. QUICK is designed to assist in the study of strategic conflicts involving a large-scale exchange of nuclear weapons. Toward this end, the system encompasses three major capabilities which are applicable to a wide range of studies: first, for a given offensive missile and bomber force and a specific set of targets, QUICK produces a detailed plan of attack which is near optimum for the conditions specified by the user. Second, it provides an expected-value estimate of the results of that attack. Finally, it produces input tapes to simulator subsystems external to QUICK.

QUICK is structured into four major subsystems: Data Management, Weapon/Target Identification, Weapon Allocation, and Sortie Generation. The principal tasks associated with each of these functional subsystems are summarized below.

a. Data Management: Assembles and reformats the target and non-target data required for a particular plan

b. Weapon/Target Identification: Selects and processes the Red and/or Blue Forces prespecified for a particular plan

c. Weapon Allocation: Allocates offensive weapons to selected targets

d. Sortie Generation: Prepares and evaluates missile and bomber attack plans.

Figure 1 displays the modules which comprise each subsystem.

Figure 2 illustrates the communication with the Central Operations Processor (COP) or executive software and the entire procedure and information flow within the QUICK system. The communication lines infer action with the COP and the integrated data base and related modules which may be executed in any reasonable order. The required processing sequence is shown by solid lines, and the information flow by broken lines.

Processing is initiated by inputting the parameters which identify the potential targets which are to be extracted from the CCTC Joint Resource Assessment Data Base (JAD) files. The COP stores selected data and dynamically conducts the proper linkage for referral by other modules within the QUICK system. Alternatively, required target data is obtained from existing updatable QUICK Data Bases and also stored and linked by the COP. Following this, specified forces are defined within the developed QUICK Data Base and processed by the Weapon/Target Identification subsystem, resulting in a Game Data Base which reflects the selected forces and targets.

The next step is to prepare an attack plan for the opposing forces. This consists of a force allocation by the Weapon Allocation Subsystem and a detailed set of attack plans prepared by the Sortie Generation subsystem.

The major inputs required to initiate this phase of processing are:

a. A game data base prepared by the Data Management and Weapon/Target Identification subsystems

b. A set of parameters which relate to the strategy associated with the plan which is to be developed

These parameters are supplied by the planner. They reflect his views as to the strategic attack objective, in terms of the relative values of the various targets being considered, the forces to be withheld, the targeting constraints to be observed, and the initiating force, i.e., which side attacks first.

The target values which are computed on the basis of these parameters reflect in a very significant way the major strategic objectives of the resultant war plan. These values are relative values and are partially contained in the data base itself. QUICK has 15 specific classes of targets. The relative values of the targets contained in any one class are included in the data base: the strategic objectives of the planner

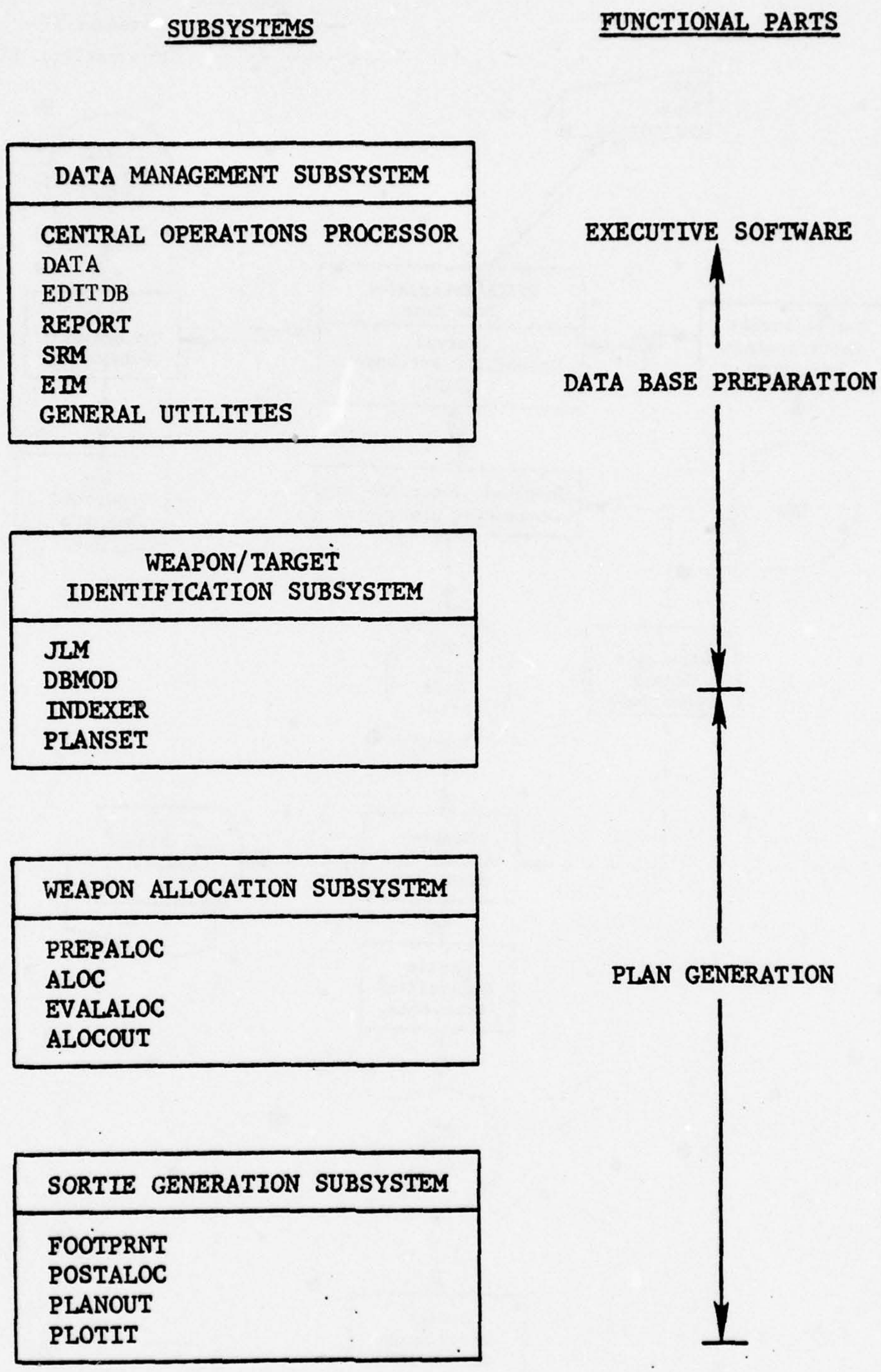



Figure 1. Major Subsystems of the QUICK System

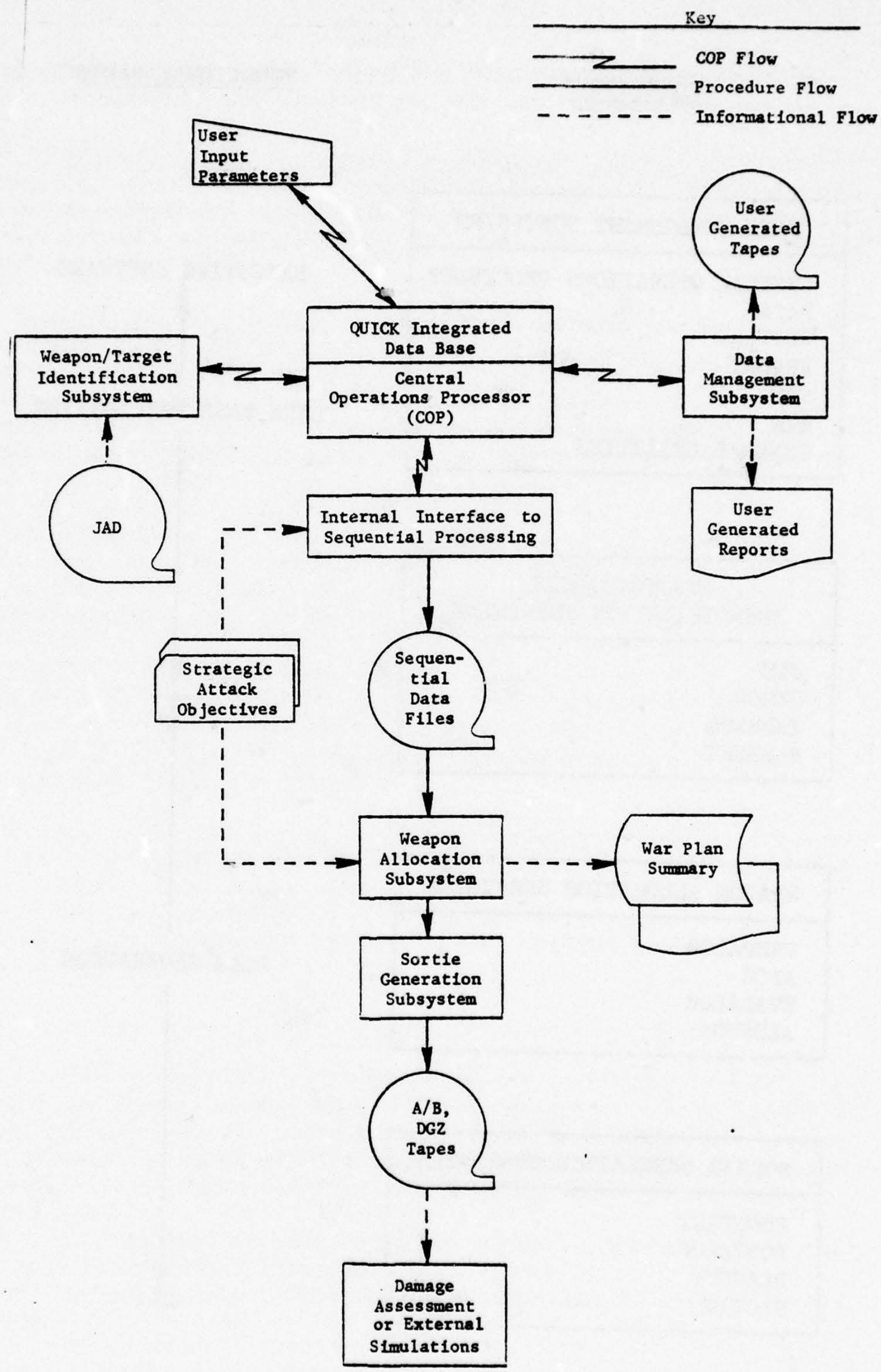



Figure 2. Procedure and Information Flow in QUICK/HIS 6000

who wants to use the plan generation function are expressed in how the value scales of these various classes of targets are related to one another. The user thereby puts more or less relative importance on each of the classes of targets in accomplishing the strategic objectives that he chooses. This, of course, will be related to the kind of strategy he is contemplating for the particular war game, whether a first or second strike, and so forth.

Having established a value for each target, the plan generation phase mathematically allocates the weapons (e.g., Red weapons to Blue targets) and prepares the detailed missile and bomber attack plans. If desired, the plans may be printed, inspected, and altered by changing the attack objectives and repeating the process. The series of Red missile and bomber events corresponding to the sortie plan is prepared in a form suitable for input to external simulators. As a user option, a war plan summary is provided which includes an expected-value estimate of the results of the attack. In addition, the aim point, Desired Ground Zero (DGZ), for each planned weapon can be output for subsequent evaluation utilizing an external damage assessment system.

While the system can proceed automatically through all steps if desired, it may be halted at the end of each module (or program), and the available output inspected for correctness and adequacy. There are standard outputs from each module and, also, mechanisms exist for producing generalized outputs (either tape/disk or printed reports). Through the COP, users may request output in any format of data items that reside within the integrated data base. The user has complete flexibility in displaying needed reports.

### 1.3 Data Management Subsystem Overview

The Data Management subsystem consists of the executive software (called COP) which maintains overall control of the entire system plus modules necessary to create, change, maintain, or display the QUICK data base. Also included is a utility package (sometimes called library) which consists of subroutines and functions which perform a variety of support tasks common to several system programs. General descriptions of these functions and their organization in subsequent sections of this volume are provided in the following subsections.

1.3.1 QUICK Integrated Data Base. QUICK employs the HIS Integrated Data Store (IDS) system to maintain and control the data base. This type of data base has all record types interconnected in some logical fashion such that data items may be queried in a generalized manner.

1.3.2 QUICK Generalized Text English Commands. The user directs the execution of QUICK through text english commands which are imperative sentences that provide meaning to the executive software. These commands permit data construction, access, maintenance, validation and display, as well as provide QUICK module executions. These commands

are user directed and, hence, are fully described within the Users Manuals. Discussions are imbedded within each module discussion.

1.3.3 QUICK System Central Operations Processor (COP). All authority for directing the QUICK system resides within the COP. This includes all interfaces with the system and the user, as well as all flow between the data base and related modules. COP also controls the execution of all modules.

1.3.4 Data Management Modules. The Data Management modules consist of modules DATA, EDITDB, REPORT, SRM, and EIM. These modules and supporting subroutines create, maintain, and display the data base. Their order of execution is not rigid. A sequence of execution holds only for the obvious situations such as a display request is possible only after the requested display items have been created and stored within the data base.

Note that generally most computer activities refer to executions in terms of programs. The compatible computer program within the QUICK system is the COP. Modules, or a set of subroutines necessary to perform some function, are executed by the COP.

1.3.5 General Utilities. The general utilities consist of subroutines and functions which perform a variety of tasks throughout the QUICK system. Among the tasks performed by the general utilities are, for example, error diagnostics, supporting position and distance calculation functions, tallying of targets, etc.

The remainder of this volume consists of three appendixes. Appendix A contains a description of the common blocks associated with data linkage between the COP and related modules. Appendix B contains the Job Control Language (JCL) required to maintain the QUICK system. Appendix C details program PERFORM whose function is to build job streams; or more appropriately to generate JCL card images.

## 1.4 Computer Software Environment and Programming Language

The QUICK system runs under control of the HIS 6000 GCOS (General Comprehensive Operating Supervisor). Except for a few utility·type subroutines which are written in the Generalized Macro Assembler Program (GMAP) language of the HIS 6000 computer system, all QUICK programs are programmed in the FORmula TRANSlator (FORTRAN) computer language.

FORTRAN is the coding language. The data base is created and maintained using the HIS Integrated Data Store (IDS) subsystem. All data items necessary for a complete QUICK system execution reside within the IDS structure.

## 1.5 Equipment Environment

The QUICK system is operational on the CCTC HIS 6000 computer system. Available to the QUICK system in addition to standard peripheral equipment are three 6000 processing modules, four 6000 system controllers, six 64K memory modules, and two seven-channel and 14 nine-channel magnetic tape handlers. Type 181 and 191 disk storage units are available for permanent files.

Types of available remote access devices relevant to the interactive capability are:

a. VIP 786W - CRT Subsystem (EIA or 188C)
b. KSR 33B - Teletype 188C (TTY)
c. RLP 300 - Remote Printer (EIA or 188C)
d. IBM 2741 - Communications terminal
e. VIP 7705 - CRT Subsystem (EIA or 188C)

## 1.6 Organization of Maintenance Manual, Volume I

Section 2 outlines the QUICK integrated data base and section 9 explains all utility subroutines and functions. Each remaining major section of this manual details a module along with the associated subroutines and functions. Major subsections for module discussion are:

a. Module input - details what chains (that is, data records) must be created prior to module execution

b. Module output - details what chains will be updated by each module

c. Functional description - details the macro function of each module and the associated major subroutines

d. Common blocks - details the contents of all internal common blocks. All common blocks used to communicate with the COP are given in appendix A. These are: C10, C15, C20, C30, C40, C50, ERRCOM, INS, IPQT, OOPS, STRING

e. Subroutine description - details the inner working of each subroutine in terms of techniques employed.

Within the QUICK system the COP (section 3) is viewed as the operating program. Based on user direction, the COP will execute overlay links or modules which perform the objectives of the user requests. Each overlay link is called through knowledge of the command verb and within each link the first subroutine is called ENTMOD (for entry module). That is, there are as many subroutines called ENTMOD as there are modules. Confusion is avoided by executing the correct overlay link. Subroutines discussion, then, always is initiated with ENTMOD whose meaning, or function, varies according to the overlay link.

Preceding Page Blank - NOT FILMED

# SECTION 2. INTEGRATED DATA BASE

## 2.1 Purpose

The integrated data base, which permits all defined records to be dynamically linked together in some logical fashion, provides the QUICK system with all necessary data. Through various entry points in the COP, the QUICK system modules retrieve data from the data base, store new data, update existing information, and delete data records which are no longer desired. In addition, the instructions generated by the text english input commands are retrieved from the integrated data base through a COP entry point.

## 2.2 Concept of Operation

The QUICK integrated data base is organized under the HIS 6000 Integrated Data Store (IDS) concept. This technique is briefly discussed below, particularly as it applies to the QUICK system. For further information on the IDS technique, please consult HIS publication DC53A I-D-S/I USER'S GUIDE.

2.2.1 IDS Records. An IDS record is a combination of user defined data, structured information and pointers which link each record to other records. The data base is divided into groups of records called 'pages.' Each page may contain up to 63 records called 'lines.' A line is a data record in the normal sense. However, an IDS line contains more information than just the data attribute values supplied by the user. Figure 3 shows a typical IDS line. As shown, the line begins with informational data such as line number, record type (assigned by the maintenance programmer as a unique identifier for each record type to distinguish it from other record types which are different in organization or content), and size of the record in characters. Following this informational data is an optional chain field used only for CALC records (see below). Next is the data attribute values. Finally, the record will contain a number of chain fields (see below).

2.2.2 Reference Code. Each IDS line (record) has a unique identifier known as its 'Reference Code.' This code is simply the page on which the line is found combined with the line number. These two numbers are formed into a single entity by multiplying the page number by 64 and adding the line number. This unique identification allows any system using the IDS technique to quickly retrieve any record. It also allows for the relating of a line to any number of other lines through the 'chain' concept which uses these reference codes to point to other lines.

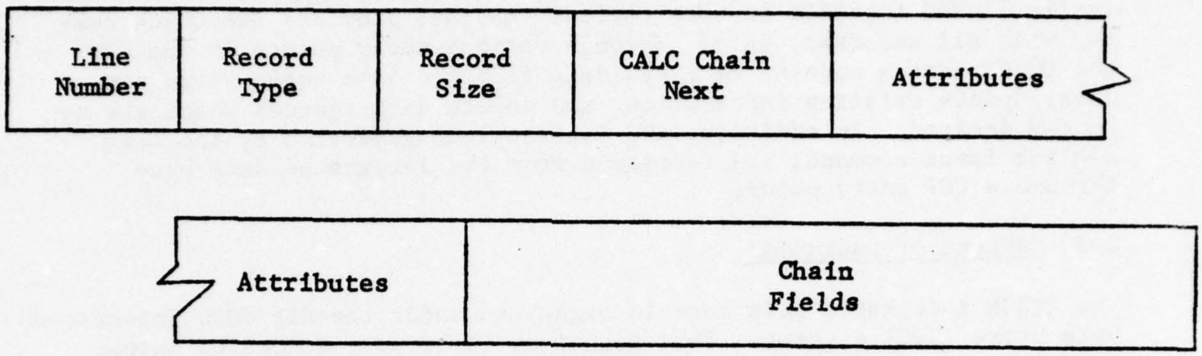



Figure 3. Integrated Data Line Format

2.2.3 IDS Chains. The data in the QUICK integrated data base easily resolves itself into what is known as an hierarchical structure. This means that there are various levels of information relating to the same subject. Note, for example, figure 4. In this figure the record referred to as "Penetration Corridor Header" would contain information that would be standard for all penetration corridors. At the next level in the hierarchy one would find the information unique to each individual corridor. Finally, at the lowest level, one would find the information needed to define each individual leg of each individual corridor. It would be most efficient to organize this data in the same manner, that is, have one record at the first level, have this record somehow related to a number of records at the second level and, finally, have each of the second level records related to several third level records. This organization is accomplished through the IDS concept of the chain.

A chain is a sequence of records each of which points to the next in the sequence until the sequence forms a closed loop. One record in the chain is called the master. The rest are called details. The master record is at the next highest level of the hierarchy from the detail, thus in figure 4, the corridor header record would be master of a chain containing, as details, the corridor records. Each of these, in turn, would be master of a chain of legs. Each line in the chain contains the information necessary to find the next line in the chain and may contain pointers to the chain's master and to the prior record of the chain.

This linkage is accomplished through the use of the chain field mentioned in subsection 2.2.1. For each chain of which the line is a master or detail, one to three chain fields will be found in the line depending upon whether the chain is to be retrieved sequentially (next), reverse sequentially (prior) and whether each detail is to have a pointer to the chain's master. At compile time, when the various record types are defined to IDS, a series of tables is set up by which each chain field in each record type is defined. When a record is created, the appropriate fields are filled with the reference codes (subsection 2.2.2) of the line which is either 'next,' 'prior' and/or 'master.' An example of this chain field concept appears in figure 5.

2.2.4 IDS Record Classes. Within the IDS concept, three classes of records or lines exist. This is due to the fact that the system will use records in a different fashion and thus, one type of definition will not necessarily be sufficient. The three types are: Primary, Secondary, and CALC.

2.2.4.1 Primary Records. A primary record is one that will be retrieved primarily by its reference code. In the QUICK system, this type of record is referred to as a header. Headers are used as entry points

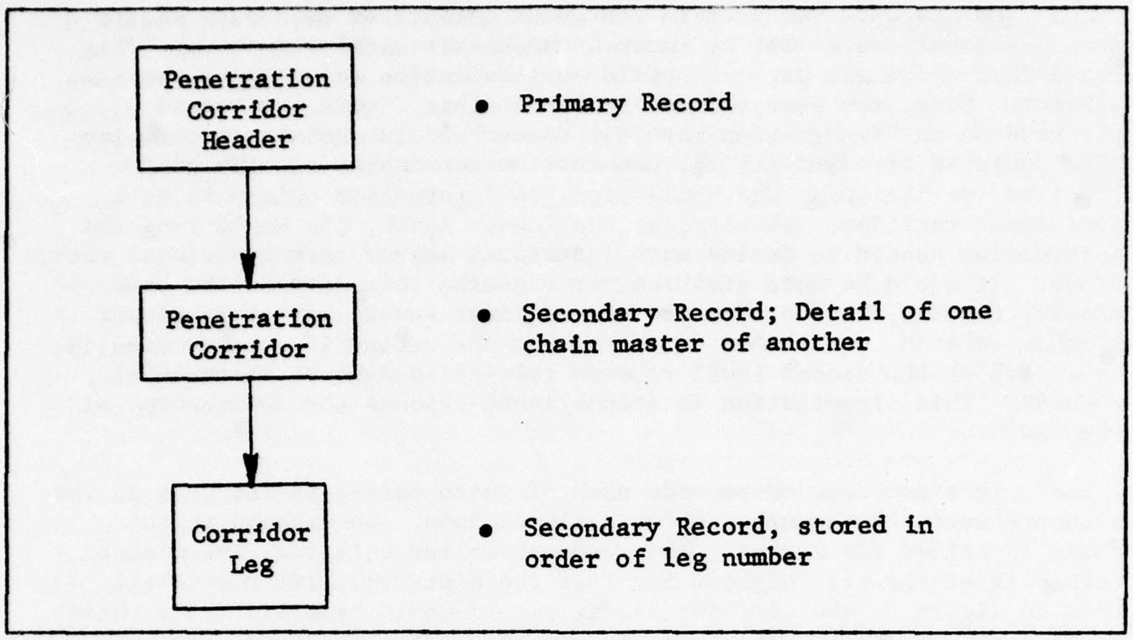



Figure 4. Example of Hierarchically Structured Data

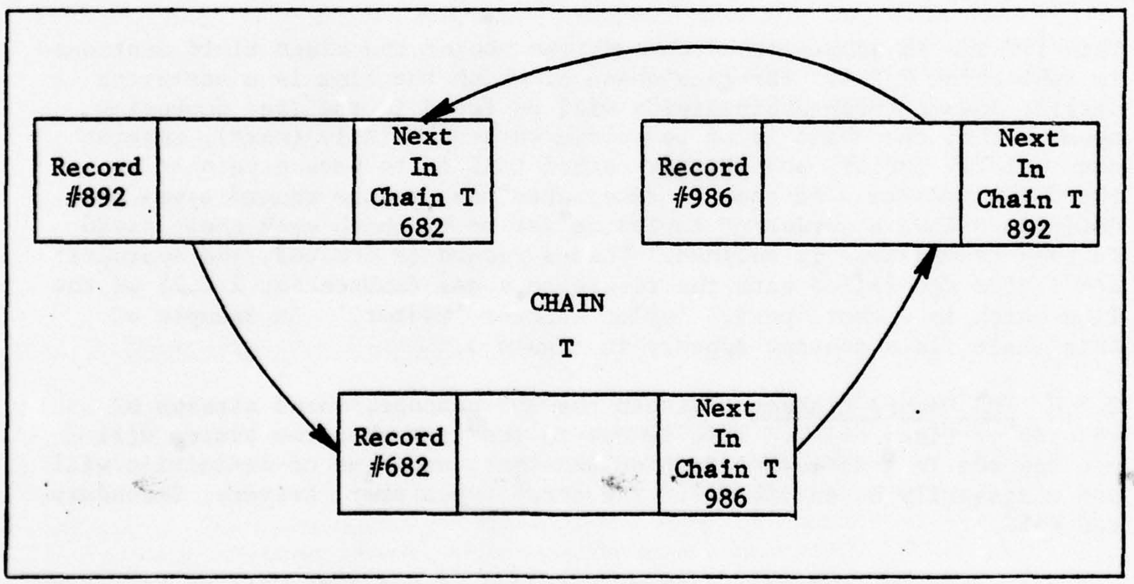



Figure 5. Record Linkage Through Chains

into the data base in the sense that by using the appropriate header, the desired data is more easily retrieved. As will be seen from an examination of the data base structure (section 2.4), virtually all scenario data is related to all other data however remotely, but it is best to use the header which is most closely related to the type of data which is desired. In figure 4 for example, if penetration corridor data is desired the penetration corridor header should be retrieved first.

2.2.4.2 Secondary Records. A secondary record is one that will be retrieved primarily as the detail of a chain. The majority of the record types in the QUICK system are secondary records. In figure 4 the corridor data would be retrieved by directly retrieving the primary header record and then traversing the chain to retrieve the secondary corridor records.

2.2.4.3 CALC Record. A CALC record is one that will be retrieved primarily via a unique identifier. This identifier is used through a hash code technique to randomly identify the page on which the record resides. From here, the CALC chain field (see figure 3) connects all CALC records for that page. The CALC record has a further advantage in that unlike a primary record, the system need not know its reference code to retrieve it, only the proper identifier. (The primary record's advantage is that once the reference code is known, retrieval is more efficient.)

In the QUICK integrated data base, CALC records are used in two ways. First, each individual target record is retrievable via its target designator code (DESIG). This allows a kind of inside out type of processing where instead of searching through all possible target classes, types, and individual examples of some to find the desired target, the target may be retrieved directly and then various chains masters retrieved to access the data further up in the hierarchy. Second, two "headers" are CALC rather than primary records. One of these is the utility table header and the other is the data organization index header. The latter is the data entry point which is used to find the reference codes of all other headers.

2.2.5 Record and Chain Tables. As records are retrieved and chains processed, two different kinds of internal tables are constantly updated. The first of these is a record table. Each record type has its own record table which, among other things, keeps track of the reference code of the last record of this type retrieved. This record is referred to as the "current" record. The table is only changed when another record of the exact same type is retrieved.

The second table is called the chain table. Each chain in the data base has its own table which contains reference codes for the last record retrieved on the chain (chain current), the record it points to and, in some cases, the chain's master and chain prior. (Because of

the structure of the QUICK data base, the chain prior will always exist). Any time, either the chain's master or any of its detail record types is retrieved (regardless of whether the chain itself was used for the retrieval), the record retrieved becomes the chain current, chain next is updated and, if possible, the master and prior are also updated. If the master and prior cannot be determined they are set to zero.

2.2.6 Chain Order. At compile time, the programmer also specifies an order for each chain. This order is merely an instruction as to where to place any new record in regard to the old ones already on the chain. The options are: 'First,' 'Last,' 'Before,' 'After' and 'Sorted.' In a chain whose order is 'first,' each new record is linked immediately after the master. In a chain whose order is 'last,' each new record is linked immediately before the master. In a chain whose order is 'after,' each new record is linked immediately after the current record. In a chain whose order is 'before,' each new record is linked immediately before the current record. Finally, in a chain whose order is 'sorted' each new record is linked such that the chain is sorted on attributes specified as sort keys.

2.2.7 Match Keys. Ordinarily when a record is placed on a chain, the storing process assumes that the desired chain master is that of the chain's current record. However, the programmer may specify that a specific master be found before a record is stored by designating a particular attribute in the master record type as a 'match-key.' For an attribute to qualify, the master record must be on a chain where the chain is sorted on the attribute in question. Then, when a record is stored as a detail to this master record, the value specified for the match-key attribute is used to retrieve the proper master for appropriate linkage.

Another property of match-keys is that they allow the QUICK system to remove a record from one chain and place it on another. This is done by altering the value of the chain master's match-key and modifying the record.

## 2.3 Functional Description

The integrated data base is accessed through various entry points of the COP. Each entry point performs a specific function related to either IDS processing, header creation and identification or input instruction code processing. The entry points to COP are summarized in table 1. Throughout the following narrative, references will be made to the COPs common blocks which are detailed in appendix A.

2.3.1 Entry Points for IDS Processing. Each of the entry points in this section performs a specific IDS function such as opening and closing data files, creating and/or deleting records, and updating created data items. The common block most often effected by these

Table 1. COP Entry Points (Part 1 of 2)

| ENTRY POINT NAME | ARGUMENTS | DESCRIPTION |
|---|---|---|
| CLZIDS | (none) | Close IDS File |
| DIRECT | (none) | Retrieves array IDS record based on its binary reference code stored in common C10 |
| DLETE | Record Type Name | Deletes current record of type named |
| HDFND | BCD reference Code, CLASS value, SIDE value, Record Type Name | Finds BCD reference code, given values for CLASS, SIDE and/or Record Type Name |
| HDPUT | BCD reference Code, CLASS value, SIDE value, Record Type Name | Adds New Header of Type named with given values for CLASS and SIDE |
| HEAD | Chain Name | Retrieves master of chain named |
| INSDEL | (none) | Deletes all input tables |
| INSFLS | (none) | Assures that any additions to input tables are recorded |
| INSGET | Array to contain output, Index of first item to retrieve, Number of items to retrieve | Obtains input instructions from input tables |
| INSPUT | Array which contains items to be inserted, Index to input table (should be set to start point minus one, will be returned as end point), Number of items to add | Inserts items in input tables |
| MODFY | Record Type Name | Modifies current record of type named to reflect current values in common |
| NEXTTT | Chain Name | Retrieves next record of chain named |

Table 1. (Part 2 of 2)

| ENTRY POINT NAME | ARGUMENTS | DESCRIPTION |
|---|---|---|
| OPNIDS | (none) | Opens IDS file |
| RETRV | Record Type Name | Retrieve record of type named (should be used only for primary or CALC records) |
| STORE | Record Type Name | Store new record of type named |

entry points is C30. Following each call that affects a particular record its binary reference code is stored in word 1 of C10 and its record type number is stored in word 4 of C10.

2.3.1.1 Entry OPNIDS. This is a one time call to open the IDS data file for processing. This entry point may only be called once in an activity.

2.3.1.2 Entry CLZIDS. This entry point closes the IDS data file. Once the data file is closed no further IDS processing may take place.

2.3.1.3 Entry DIRECT. Any individual data record may be retrieved by this entry point through its reference code. Prior to the call to DIRECT, the calling routine must store the binary reference code (page number * 64 + line number) in word 1 of common block C10. Following the call, the appropriate attributes in common C30 will have been set to the values of the desired record.

2.3.1.4 Entry NEXTTT. This entry point retrieves the next record of a specified chain which is identified through the argument. Following the call, the appropriate attributes in C30 will have been set to the values of the record retrieved. An end-of-chain condition is met if the master of the chain has been retrieved. The user must check word 4 of common C10 against the master's record type number for end condition.

2.3.1.5 Entry HEAD. This entry point retrieves the master record of a chain as specified within the calling argument. Following the call, the appropriate attributes in common C30 will have been set to the values of the record retrieved. Also, after this call, all chaining (that is calls to NEXTTT) will begin with the first record within the chain.

2.3.1.6 Entry STORE. By specifying the record type's alphabetic name, this entry point creates the record of the type given. Prior to the call the appropriate attributes in common C30 must have been set to the desired values.

2.3.1.7 Entry RETRV. This entry point retrieves primary or CALC records. The record to be retrieved is identified by the record type alphabetic name. Before the call to retrieve a primary record, common C15 must contain the BCD representation of the desired record reference code. Before the call to retrieve a CALC record, attribute DESIG must be set. Following the call, the appropriate attributes in common C30 will have been set to the value of the record retrieved.

2.3.1.8 Entry MODFY. This entry point modifies the current record of the type specified through the argument. Before calling MODFY, the record to be modified must be the current record of that type. Also, all attributes which are to be modified must be set to the new values. In addition, be aware that if a match key is modified, the current record will be moved to a new chain based on the new match key value.

2.3.1.9 Entry DLETE. This entry point deletes the current record of the type specified through the argument. Before calling DLETE, the record to be deleted must be the current record of that type.

2.3.1.10 Error Recovery. When an error occurs during IDS processing, an error code is inserted in word 6 of common block C10 and subroutine ERPROC is called (see section 3). Calling routines may control the action which ERPROC will take by altering the entries in common block ERRCOM. The action codes are:

ABORT - Core dump occurs and processing stops. (This is the default for NORMAC)

FLAG - Processing continues for this sentence but error flag (common block OOPS) is set. (This is the default for CHCKAC)

PASS - Processing continues -- no action is taken.

Either of the action codes (NORMAC or CHCKAC) may be changed. Error codes may be added to the list for CHCKAC (CHEKS, the list, is originally empty). The report code on which the ERPROC message appears may also be changed. If the contents of ERRCOM are altered they are not reset by COP.

2.3.2 Entry Points for Header Creation and Identification. The two entry points in this section enter headers into the data base and allow calling programs to obtain the BCD reference code of a header for use with a call to RETRV. Each entry has the same four parameters:

1 - BCD reference code (Type CHARACTER*8)
2 - Value for CLASS attribute
3 - Value for SIDE attribute
4 - Record Type Name

Note: The first parameter should always be checked after the call as a value of '00000000' indicates an error has occurred.

2.3.2.1 Entry HDPUT. This entry point creates a header and saves its BCD reference code in the data organization index. Either or both values for CLASS or SIDE may be blank.

2.3.2.2. Entry HDFND. This entry point retrieves the BCD reference code of a header for use in a call to RETRV. The BCD code retrieved will be that of a header whose values match those set in parameters 2, 3, and 4. If any of these parameters is blank (including Record Type Name) it is not checked. Note that if both CLASS and Record Type Name are blank the code returned is unpredictable.

2.3.3 Input Instruction Code Processing. Previous subsections identified the developed QUICK IDS data base in terms of record structure and content of each record. This subsection identifies how user input commands are constructed for interrogation by any module.

Following syntax analysis, the COP transforms the text English input, now in symbolic form, into a series of executable instruction codes. These codes are stored in utility tables which are, logically, an array which may be moved into a subroutine's local array by specifying the local array, starting index, and number of items to be moved. Similarly, a calling subroutine may alter the instruction array.

Information stored within the utility tables parallels the command syntax. That is, verb information is initially given, followed by clause related data, and further subsetted into phrases associated with each clause. The discussion to follow is similarly arranged.

2.3.3.1 Verb and Adverb Instruction. The first word of the array contains the identifying number of the verb, the second word contains the number of adverbs which modify the verb (see table 2). If the second word is zero there is no further input. If the second word is nonzero, pairs of words follow for each adverb modifying the verb. The first word of each pair contains the adverb identifying number followed by either a zero, if the adverb introduces no clause, or the array number into the instruction array where the first instruction of each clause is found.

Each clause introduced by an adverb (see table 2) contains a series of instructions beginning in the indicated word and continuing sequentially until an 'end of clause' instruction is encountered. The type of instructions contained in the clause vary depending on which type is involved.

2.3.3.2 Input Clause Types. Three types of clauses may be found in the input instruction table which are distinguished by the type of phrases that are included and the manner in which they are connected. The three types are: boolean, sequential and elemental.

a. Boolean Clauses -- The boolean clause is the highest level of organization. Any routine which processes a boolean clause need only execute each command as encountered and this result will be the final value of a logical variable. Following any mathematical calculations, a boolean clause will be a series of logical operations and relational comparisons. The logical operations will involve logical variables which must be set up by the routine executing the instructions. The variables are identified by an index number to be used in a single logical array in the executing routine. The relational comparisons will be in code as three per operation. First there

Table 2. Verb and Adverb Array Structure

| ARRAY NUMBER | DESCRIPTION |
|---|---|
| 1 | Verb number (see User Manual Volume I) |
| 2 | Number of adverbs |
| 3 | First adverb number (see Users Manual Volume I) |
| 4 | Array number of first instruction in clause (say, for example I) or zero if no clause |
| 5 | Second adverb number |
| 6 | Array number (say J) of first instruction in clause |
| ⋮ | ⋮ |
| I | First instruction for first clause. Adverb is defined in array number 2 |
| ⋮ | ⋮ |
| I+i | ='1', instruction code indicating end of clause |
| J | First instruction for second clause |
| ⋮ | ⋮ |
| J+i | ='1' |
| ⋮ | ⋮ |

will be an instruction to 'load' a particular value (attribute, constant, etc.); second an instruction will specify the type of comparison (i.e., equal to, greater than) and the value to which the comparison is made. Finally, an instruction will specify that the logical result of the comparison be stored in a logical variable. In the second step, the comparison may be made to an internal variable which, similarly to the logical variable, is set up as an array in the executing routine. This internal variable is the result of an earlier mathematical calculation.

b. Sequential Clause -- Following any mathematical calculations, a sequential clause will be a series of relational comparisons. Each comparison consists of two parts. First there will be an instruction to 'load' a particular value. Second an instruction will specify the type of comparison or operation and the value involved. Often in a sequential clause, the relationship 'equals' takes on the meaning 'is set to.' If the relation is generated by a simple relational phrase the one pair of instructions will be followed by an 'end of phrase' instruction code = '2'. However, complex relational phrases may contain dummy logical operations to aid in the executing routine's discrimination. If the subject and object of the phrase is a collection, a series of instruction pairs is generated, one for each individual relationship. These pairs of instructions are separated by a logical 'AND' instruction with a zero index. (For example the phrase '(LAT,LONG) = (10,20)' would symbolically generate LAT=10 AND LONG=20.) The last pair is followed by an end of phrase. Similarly, an extended equals phrase generates a series of instruction pairs separated by logical 'OR' instructions. (For example the phrase LAT=10 OR 20 symbolically generates LAT=10 OR LAT=20.) Naturally a combination of these complex forms generates a combination of the sequences. (For example (LAT,LONG) = (10,20) OR (30,40) symbolically generates LAT=10 AND LONG=20 OR LAT=30 AND LONG= 40.) In any case, an end of phrase appears only after the last instruction pair of the phrase.

c. Elemental Clause -- This is the lowest form of clause. In effect it is merely a symbol translation of the text English input without an attempt at interpretation. The executing subroutine is called upon to determine what actions are to be taken depending upon the string of symbols.

2.3.3.3 Input Instruction Codes. Each instruction is a series of words the first of which is an instruction code which contains detailed information as to the operation to be carried out. For each clause, then, several instructions are defined sequentially such as:

| ARRAY NUMBER | DESCRIPTION |
|---|---|
| I | Instruction Code |
| ⋮ | ⋮ |
| I+i | Instruction Code |
| ⋮ | ⋮ |
| I+i+j | Instruction Code |
| | ⋮ |
| | ='1', instruction code indicating end of clause |

Table 3 details the meaning of the various bit configurations that comprises the instruction code.

2.3.3.4 Input Instruction Format. The series of words begun by an instruction follows a particular format depending upon the instruction involved. These formats fall into four groups: miscellaneous instructions, logical instructions, instructions using internal variables and instructions which follow the "general instruction format." All of these various formats are summarized in table 4. Unless specified various values for addresses and numbers may be found in User Manual Volume I.

2.3.3.4.1 Miscellaneous Instructions. Each of these is begun by a particular instruction followed by a format unique to the code alone. End of clause and end of phrase (1 and 2 respectively) have no words following. Operator Follows (3) has one word following which contains the operator's identifying numbers (e.g., EQUALS is 7). A 'LIKE String Follows' (11) is generated by the LIKE relational operator and contains the information to identify the desired record. The format for a LIKE string varies depending upon whether the value for the identifying attribute is alphabetic or numeric.

A 'Long String Follows' is merely the instruction code (13) followed by the number characters in the string, followed by the string itself. Note however, that the string is formed in pairs of words so that a 13-character string would generate four words for the string itself (six words for the instruction).

Finally, a 'Special Word Follows' (14) is followed by the identifying number of the special word.

2.3.3.4.2 Logical Instructions. Each of these instructions consists of an instruction code followed by the index of a logical variable. If this index is zero, the instruction is either a dummy instruction

Table 3. Instruction Code Bit Configuration

| BITS | DESCRIPTION |
|---|---|
| 29-32 | General instruction code<br>0 = Terminator and Operator follows<br>1 = General Item Follows<br>2 = Equals<br>3 = Greater Than, or Greater Than or Equal To<br>4 = Less Than, or Less Than or Equal To<br>5 = Logical Operations<br>6 = Loads or Stores<br>7 = Adds or Subtracts<br>8 = Multiplies or Divides<br>9 = Powers<br>For general instruction code = 0, 1, or 5 |
| 33-35 | Particular Code Discriminator<br>For general instruction code = 3, 4, 6, 7, or 8 |
| 33 | Second Code Discriminator<br>0 = First operation shown above<br>1 = Second operation shown above |
| 34-35 | Value Type Discriminator<br>1 = Alphabetic value<br>2 = Numeric value<br>3 = Internal variable |

Table 4. Input Instruction Formats (Part 1 of 3)

| ARRAY NUMBER | DESCRIPTION |
|---|---|
| | **End of Clause** |
| I | ='1', instruction code |
| | **End of Phrase** |
| I | ='2', instruction code |
| | **Operator Follows** |
| I | ='3', instruction code |
| I+i | Operator number |
| | **LIKE String Follows - Alphabetic Identifier** |
| I | ='11', instruction code |
| I+1 | Identifier attribute address |
| I+2 | Identifier attribute number |
| I+3 | ='9', instruction code |
| I+4 | ='9', instruction code |
| I+5 | First half of identifying value |
| I+6 | Second half of identifying value |
| | **LIKE String Follows - Numeric Identifier** |
| I | ='11', instruction code |
| I+1 | Identifies attribute address |
| I+2 | Identifies attribute number |
| I+3 | = '10', instruction code |
| I+4 | = '10', instruction code |
| I+5 | Floating point identifying value |
| | **Long String Follows** |
| I | ='13', instruction code |
| I+1 | Number of characters in long string |
| I+2 | First six characters of first pair |
| I+3 | Second six characters of first pair |
| ⋮ | |
| I+(N-1) | First six characters of last pair |
| I+N | Second six characters of last pair |

Table 4. (Part 2 of 3)

| ARRAY NUMBER | DESCRIPTION |
|---|---|
| | **Special Word Follows** |
| I | ='14', instruction code |
| I+1 | Special word number |
| | **Logical Instructions** |
| I | Instruction code (see table 5) |
| I+1 | Index of logical variable (if this is 0 it applies to the variable or relation which follows the instruction) |
| | **Internal Variable Instructions** |
| I | Instruction code (see table 6) |
| I+1 | Index of internal variable |
| | **General Instruction Format - Alphabetic Value** |
| I | Instruction code (see table 7) |
| I+1 | '9' |
| I+2 | First half of alphabetic value |
| I+3 | Second half of alphabetic value |
| | **General Instruction Format - Numeric Value** |
| I | Instruction code (see table 7) |
| I+1 | '10' |
| I+2 | Floating point value |
| | **General Instruction Format - Attribute (no OF phrase)** |
| I | Instruction code (see table 7) |
| I+1 | '6' |
| I+2 | Attribute's address |
| I+3 | Attribute's number |
| I+4 | '0' |

Table 4. (Part 3 of 3)

| ARRAY NUMBER | DESCRIPTION |
|---|---|
| | General Instruction Format - Attribute With OF Phrase - Alphabetic Identifier |
| I | Instruction code (see table 7) |
| I+1 | '6' |
| I+2 | Attribute's address |
| I+3 | Attribute's number |
| I+4 | Identifier attribute's address |
| I+5 | Identifier attribute's number |
| I+6 | '9' |
| I+7 | First half of alphabetic value |
| I+8 | Second half of alphabetic value |
| | General Instruction Format - Attribute With OF Phrase - Numeric Identifier |
| I | Instruction code (see table 7) |
| I+1 | '6' |
| I+2 | Attribute's address |
| I+3 | Attribute's number |
| I+4 | Identifier attribute's address |
| I+5 | Identifier attribute's number |
| I+6 | '10' |
| I+7 | Floating point value |

Table 5. Input Instruction Codes for Logical Variables

| CODE NUMBER | DESCRIPTION |
|---|---|
| 41 | Load from logical variable |
| 42 | Store into logical variable |
| 43 | Logical AND with logical variable |
| 44 | Logical OR with logical variable |
| 45 | Logical NOT with logical variable |

Table 6. Input Instruction Codes for Internal Variables

| CODE NUMBER | DESCRIPTION |
|---|---|
| 19 | Equals an internal variable |
| 27 | Greater than an internal variable |
| 31 | Greater than or equal to an internal variable |
| 35 | Less than an internal variable |
| 39 | Less than or equal to an internal variable |
| 51 | Load an internal variable |
| 55 | Store into an internal variable |
| 59 | Plus an internal variable |
| 63 | Minus an internal variable |
| 67 | Times an internal variable |
| 71 | Divided by an internal variable |
| 75 | To the power of an internal variable |

Table 7. Input Instruction Codes Found in the General Format

| CODE NUMBER | DESCRIPTION |
|---|---|
| 9 | Alphabetic value follows |
| 10 | Numeric value follows |
| 17 | Equals an alphabetic value |
| 18 | Equals a numeric value |
| 25 | Greater than an alphabetic value |
| 26 | Greater than a numeric value |
| 29 | Greater than or equal to an alphabetic value |
| 30 | Greater than or equal to a numeric value |
| 33 | Less than an alphabetic value |
| 34 | Less than a numeric value |
| 37 | Less than or equal to an alphabetic value |
| 38 | Less than or equal to a numeric value |
| 49 | Load an alphabetic value |
| 50 | Load a numeric value |
| 57 | Plus an alphabetic value |
| 58 | Plus a numeric value |
| 61 | Minus an alphabetic value |
| 62 | Minus a numeric value |
| 65 | Times an alphabetic value |
| 66 | Times a numeric value |
| 69 | Divided by an alphabetic value |
| 70 | Divided by a numeric value |
| 73 | To the power of an alphabetic value |
| 74 | To the power of a numeric value |

(see section 2.3.3.2) or is a pending operation to be prefixed to the next logical instruction. (The usual case here is an AND or OR instruction followed by a NOT instruction.) The instruction codes used with this format appear in table 5.

2.3.3.4.3 Instructions Using Internal Variable. Each of these instructions consists of an instruction code followed by the index of an internal variable. The instruction codes used with this format appear in table 6. An internal variable is the storage of intermediate mathematical calculations. The instruction code informs routines where the storage resides.

2.3.3.4.4 The General Instruction Format. The majority of instructions use this format. As shown in table 4 this format has five variations and may contain from three to nine words. The first major branch occurs by checking word 2. If '9' or '10' is found a constant follows. If '6' is found the value is an attribute and must be retrieved. The other major branch occurs with attributes. If word '5' is zero the attribute is not modified. Otherwise, the information necessary to retrieve the proper record containing the desired attribute is given. This occurs as a result of an 'OF' phrase. The instruction codes which use the general instruction format are found in table 7.

2.3.3.5 Input Instructions Code Entry Points. Routines in the QUICK system may access the input instruction array through the entry points: INSGET, INSPUT, INSFLS, and INSDEL (see table 1).

2.3.3.5.1 INSGET. This entry point moves a specified portion of the input instruction array into a specified array. The arguments are the local routine's array, the index of the first word to retrieve and the number of words to retrieve.

2.3.3.5.2 INSPUT. This entry point inserts new instructions. The arguments are the local routine's array which contains the new instructions, an index which is set to the word to be filled first minus one, and the number of words to be inserted. Note that the second argument is returned with the index number of the last word filled.

2.3.3.5.3 INSFLS. This entry point assures that all updates to the instruction array are recorded. It should be called after a completed series of calls to INSPUT. It has no arguments.

2.3.3.5.4 INSDEL. This entry point deletes the input instruction table. It has no arguments.

2.3.3.6 Instruction Code Example. Through text English commands, the user has at hand a powerful method of input data generalization. The series of instruction codes which is simply a translation of the input commands, then, is likewise non-restrictive. The previous subsection outlined all of the various combinations that instruction codes could have. In

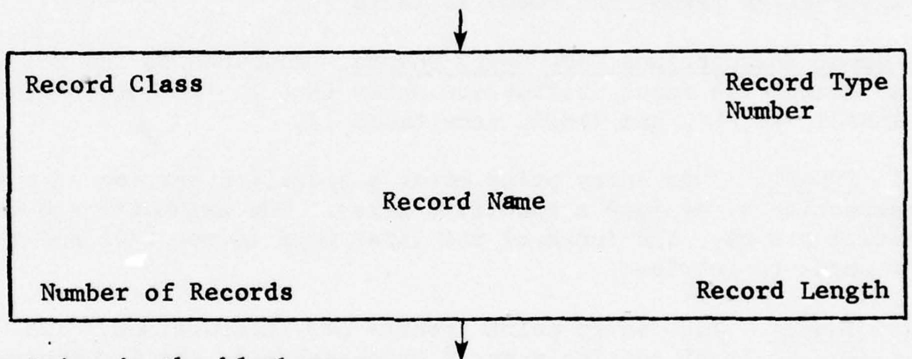
order to summarize and draw together many of the pertinent points concerning instruction code construction, table 8 is given. This example command presents a sentence of a complex nature as well as the instruction code array that would be constructed by the COP. Note that by logically scanning the array (calls to INSGET) modules may readily compute the desired result.

## 2.4 QUICK's Data Base Structure

The structured data base is centralized onto one data file. User-directed or module-generated data is properly included within the data base structure automatically by the IDS system as directed through the COP.

Modules as they are executed do not generate new data files linkage. All linkage must be initially defined prior to execution. Only the contents of data records are updated, not the structure. Figures to be presented within this subsection define the structure.

The IDS data records, as given in future subsections, are depicted with a rectangular block and has a format as shown below.




The entries in the block are:

- Record Class - Indicates how records are stored and accessed. A 'C' defines calculated (CALC) records, a 'P' Primary Records, or an 'S' for Secondary Records
- Record Type Number - Unique number assigned to the record
- Record Name - COBOL name for the record
- Number of Records - Estimated number of data records denoted by the record block
- Record Length - Number of characters requested for the block
- Double Line - Denotes CALC records

Table 8. Example of Instruction Codes (Part 1 of 5)

Command

CHANGE WHERE SIDE=BLUE AND RANGE BETWEEN RANGE OF TYPE 'B-52' AND RANGE OF TYPE 'B-52' TIMES 10 SETTING CEP LIKE TYPE 'B-52' (SPEED,SPDLO) = (10000,9000)

| ARRAY NUMBER | ARRAY VALUE | DESCRIPTION |
|---|---|---|
| 1 | 4 | Verb number - CHANGE |
| 2 | 2 | Number of Adverbs |
| 3 | 22 | Number of first Adverb - WHERE |
| 4 | 7 | Array number of beginning of WHERE clause |
| 5 | 17 | Number of second Adverb - SETTING |
| 6 | 64 | Array number of beginning of SETTING clause |
| | | Beginning of WHERE clause, first group of instructions calculates RANGE OF TYPE 'B-52' TIMES 10 |
| 7 | 59* | Load numeric value |
| 8 | 6 | Numeric is an attribute |
| 9 | 148 | Address of attribute (RANGE) |
| 10 | 125 | Number of attribute (RANGE) |
| 11 | 59 | Attribute is modified by OF phrase - this is address of identifier attribute (TYPE) |
| 12 | 51 | Number of identifier attribute (TYPE) |
| 13 | 9 | Indicates value for identifying attribute is alphabetic |
| 14 | B-52 | First half of value |
| 15 | | Second half of value (= blanks) |
| 16 | 66* | Multiply by numeric |
| 17 | 10 | Indicates numeric is a constant number |
| 18 | 10. | Numeric constant |
| 19 | 55* | Store calculated value in an internal variable |
| 20 | 1 | Index number of internal variable |

*Instruction code.

Table 8. (Part 2 of 5)

| ARRAY NUMBER | ARRAY VALUE | DESCRIPTION |
|---|---|---|
| | | The next group of instruction evaluates the phrase SIDE=BLUE |
| 21 | 49* | Load alphabetic value |
| 22 | 6 | Alphabetic is an attribute |
| 23 | 2 | Attribute's address (SIDE) |
| 24 | 2 | Attribute's number (SIDE) |
| 25 | 0 | Indicates no OF phrase |
| 26 | 17* | Equal to alphabetic? |
| 27 | 9 | Alphabetic is alpha constant |
| 28 | BLUE | First half of alpha constant |
| 29 | | Second half of alpha constant (= blanks) |
| 30 | 42* | Store result of comparison in a logical variable |
| 31 | 1 | Logical variable index number |
| | | The next group of instructions evaluates RANGE BETWEEN RANGE OF TYPE 'B-52' AND RANGE OF TYPE 'B-52' given that the final portion has already been calculated |
| 32 | 50* | Load Numeric Value |
| 33 | 6 | Numeric is an attribute |
| 34 | 148 | Attribute's address (RANGE) |
| 35 | 125 | Attribute's number (RANGE) |
| 36 | 0 | No OF phrase |
| 37 | 30* | Greater than or equal to numeric? |
| 38 | 6 | Numeric is an attribute |
| 39 | 148 | Attribute's address (RANGE) |
| 40 | 125 | Attribute's number (RANGE) |
| 41 | 59 | Address of identifier in OF phrase (TYPE) |

*Instruction code.

Table 8. (Part 3 of 5)

| ARRAY NUMBER | ARRAY VALUE | DESCRIPTION |
|---|---|---|
| 42 | 51 | Number of identifier (TYPE) |
| 43 | 9 | Value for identifier is alphabetic |
| 44 | B-52 | First half of alpha value |
| 45 | | Second half of alpha value (= blanks) |
| 46 | 43* | Logical AND indicates that result of previous comparison is to be logically 'anded' to following comparison. Thus word 47=0 |
| 47 | 0 | |
| 48 | 50* | Load Numeric Value |
| 49 | 6 | Numeric is an attribute |
| 50 | 148 | Attribute's address (RANGE) |
| 51 | 125 | Attribute's number (RANGE) |
| 52 | 0 | No OF phrase |
| 53 | 39* | Less than or equal to an internal variable? |
| 54 | 1 | Index number of internal variable |
| 55 | 42* | Store the results of the logical 'and' of the two comparisons in a logical variable |
| 56 | 2 | Index number of logical variable |
| | | Now the previously set logical variables are used to evaluate the WHERE clause |
| 57 | 41* | Load value of logical variable |
| 58 | 1 | Index number of logical variable |
| 59 | 43* | AND to value of logical variable |
| 60 | 2 | Index number of logical variable |
| 61 | 42* | Store result in logical variable |
| 62 | 3 | Index number of logical variable |
| 63 | 1* | End of clause |

*Instruction code.

Table 8. (Part 4 of 5)

| ARRAY NUMBER | ARRAY VALUE | DESCRIPTION |
|---|---|---|
| | | Beginning of SETTING clause. First group of instructions is for CEP LIKE TYPE TYPE 'B-52' |
| 64 | 50* | Load Numeric |
| 65 | 6 | Numeric is an attribute |
| 66 | 145 | Attribute's address (CEP) |
| 67 | 122 | Attribute's number (CEP) |
| 68 | 0 | No OF phrase |
| 69 | 11* | LIKE string follows - Indicates that loaded attribute should be set to the value of the same attribute in identified record |
| 70 | 59 | Identifier attribute's address (TYPE) |
| 71 | 51 | Identifier attribute's number (TYPE) |
| 72 | 9 | Identifier's value is alphabetic |
| 73 | 9 | Identifier's value is alphabetic |
| 74 | B-52 | First half of alphabetic value |
| 75 | | Second half of alphabetic value (= blanks) |
| 76 | 2* | End of Phrase |
| | | The Last group of instruction is for (SPEED,SPDLO)=1000,900) |
| 77 | 50* | Load Numeric |
| 78 | 6 | Numeric is an attribute |
| 79 | 146 | Attribute's address (SPEED) |
| 80 | 123 | Attribute's number (SPEED) |
| 81 | 0 | No OF phrase |
| 82 | 18* | Set equal to numeric |
| 83 | 10 | Numeric is a constant |
| 84 | 1000. | Numeric constant |
| 85 | 43* | Logical AND used as phrase connector |

*Instruction code.

Table 8. (Part 5 of 5)

| ARRAY NUMBER | ARRAY VALUE | DESCRIPTION |
|---|---|---|
| 86 | 0 | — |
| 87 | 50* | Load Numeric |
| 88 | 6 | Numeric is an attribute |
| 89 | 157 | Attribute's address (SPDLO) |
| 90 | 141 | Attribute's number (SPDLO) |
| 91 | 0 | No OF phrase |
| 92 | 18* | Set equal to numeric |
| 93 | 10 | Numeric is a constant |
| 94 | 900. | Numeric constant |
| 95 | 2* | End of phrase |
| 96 | 1* | End of clause |

*Instruction code.

Individual record blocks are connected with vectors and arrows representing each chain relationship in the file. Beside each chain vector, the chain name is entered. Below each chain name an indication is given to describe how records are positioned or entered in the chain. The appropriate entries are: 'F' for First, 'L' for Last, 'S' for Sorted, 'B' for Before, and 'A' for After.

Organizationally, the QUICK integrated data base may be divided into two parts: the scenario (or gaming) data and the organization data.

2.4.1 Scenario Data Structure. Figure 6 is a picture of the scenario data structure. However, as this structure is quite complex, it will be divided into three parts for discussion purposes. The figures given for the subdivisions are incomplete in that they do not have connected to them the chains which interrelate the three subdivisions.

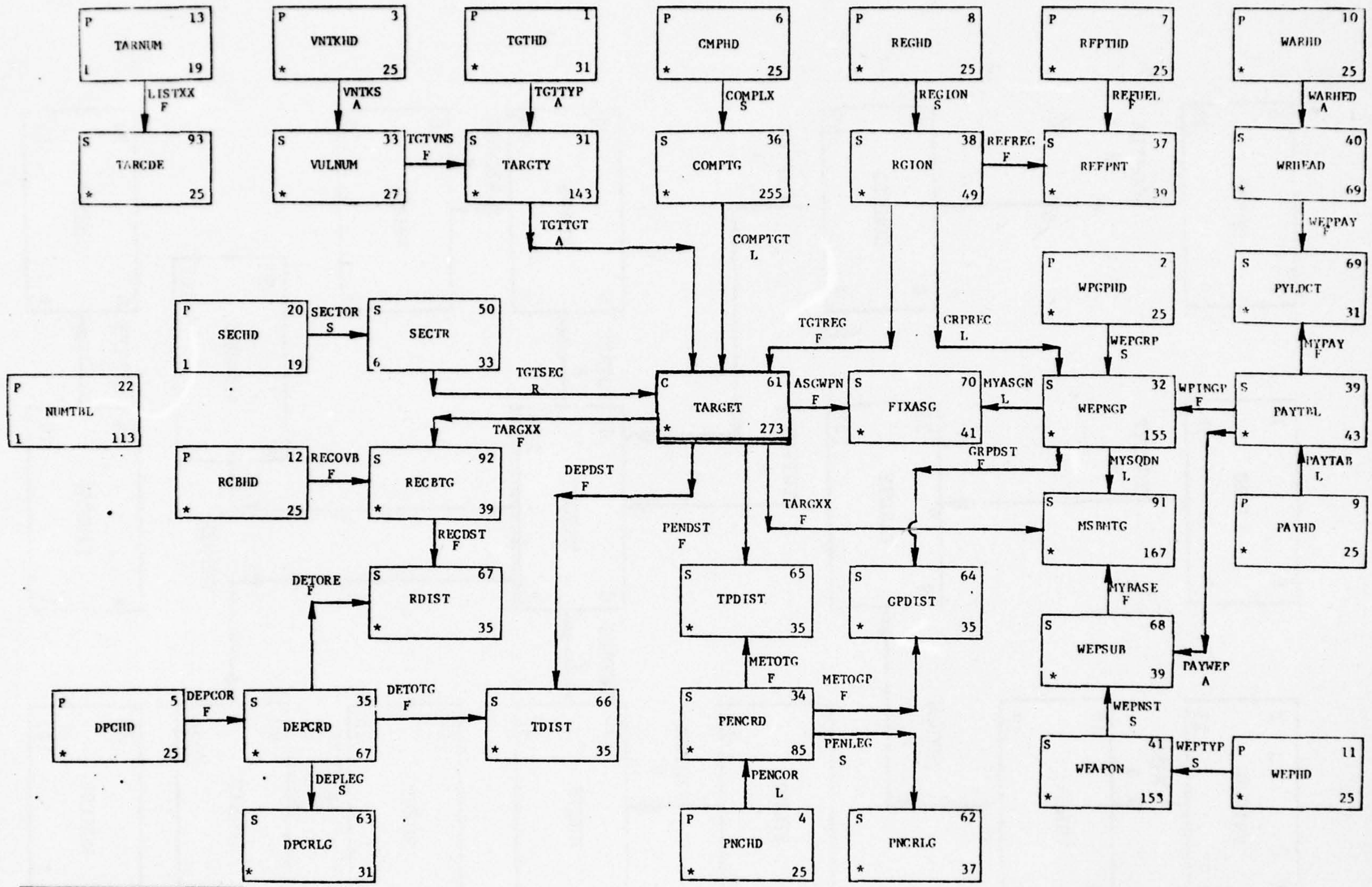
2.4.1.1 Target Data Structure. Figure 7 shows the target data organization. The TARGET record is the central record type of the data base and is a CALC record. The principal hierarchy is that of target class header (TGTHD) target type (TARGTY) and individual target (TARGET). Target types are also grouped by vulnerability number. Targets are grouped by sector, region and complex. The TARGXX chain links some individual target records to additional data. In one case the data is that for a recovery base (RECBTG). In the other case the target is also a missile, bomber, or tanker base (MSBMTG).

Figure 7 also shows two structures which are not connected to the rest. One is a table of various scenario quantities (NUMTBL). The other is a list of target reference codes (TARNUM-TARCDE) sorted according to module ALOC's needs.

2.4.1.2 Weapon Data Structure. Figure 8 shows the weapon data structure. One hierarchy contains the weapon class (WEPHD), weapon type (WEAPON), weapon type subdivided by payload (WEPSUB) and the individual weapon base (MSBMTG). There is also a warhead class (WARHD) with warhead type as details (WRHEAD). Weapons are connected to their warheads via a payload table (PAYTBL) which is master of one chain containing all weapon subtypes (WEPSUB) which utilize that table and another chain which contains counts (PYLDCT) of the various warhead types.

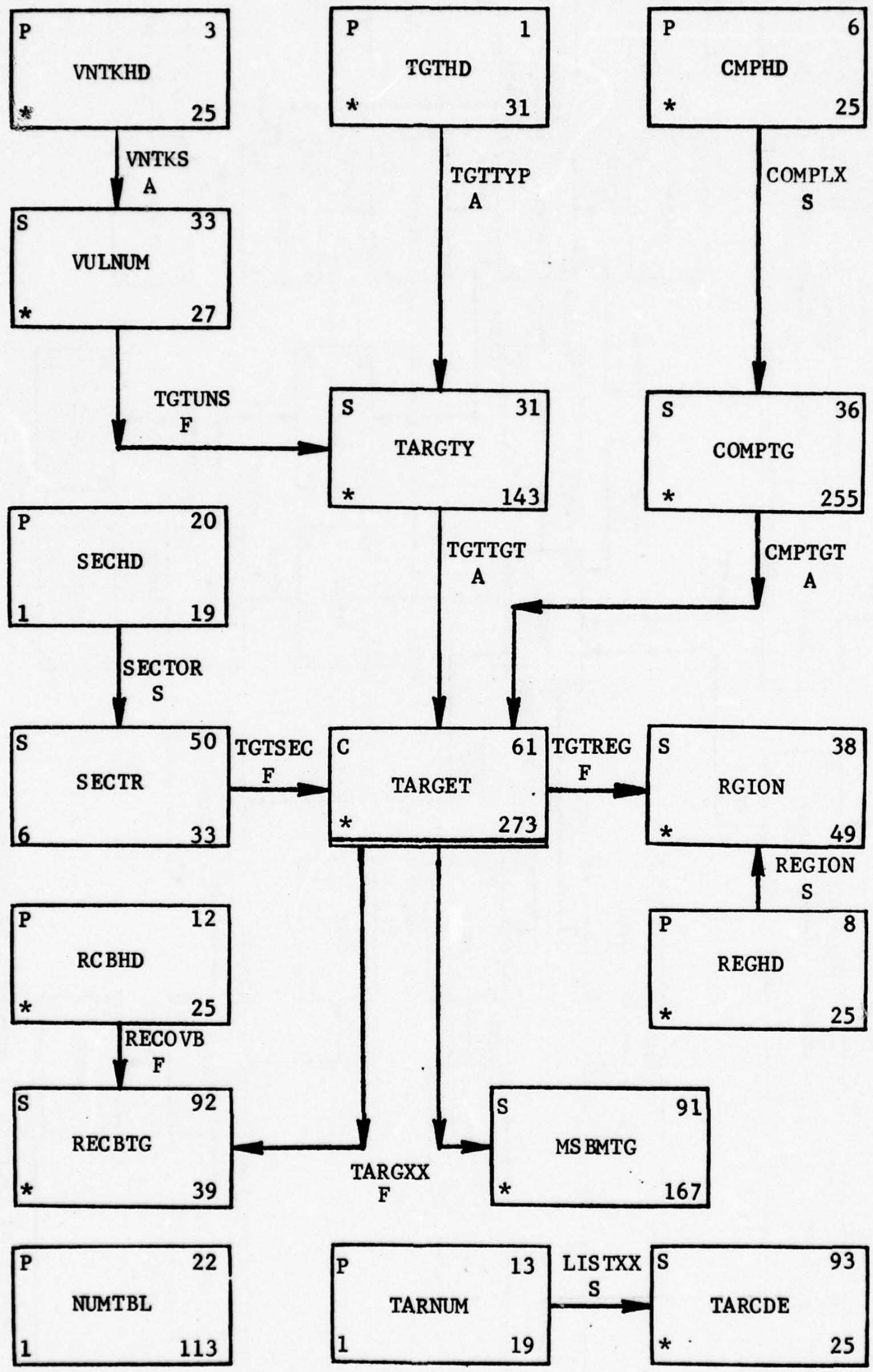
Finally, the QUICK system creates weapon groups (WEPNGP) which has a payload table and a number of individual bases (MSBMTG). Finally, the weapon group may be fix assigned (FIXASG) to a target or targets.

2.4.1.3 Geographic Data Structure. Figure 9 shows the geographic data. The principal items here are the two corridor hierarchys. Penetration corridor class (PNCHD) with individual corridors (PNCRD) and their individual legs (PNCRLG). Depenetration corridor class (DPCRLG) with corridors (DEPCRD) and legs (DPCRLG). Corridors are connected to



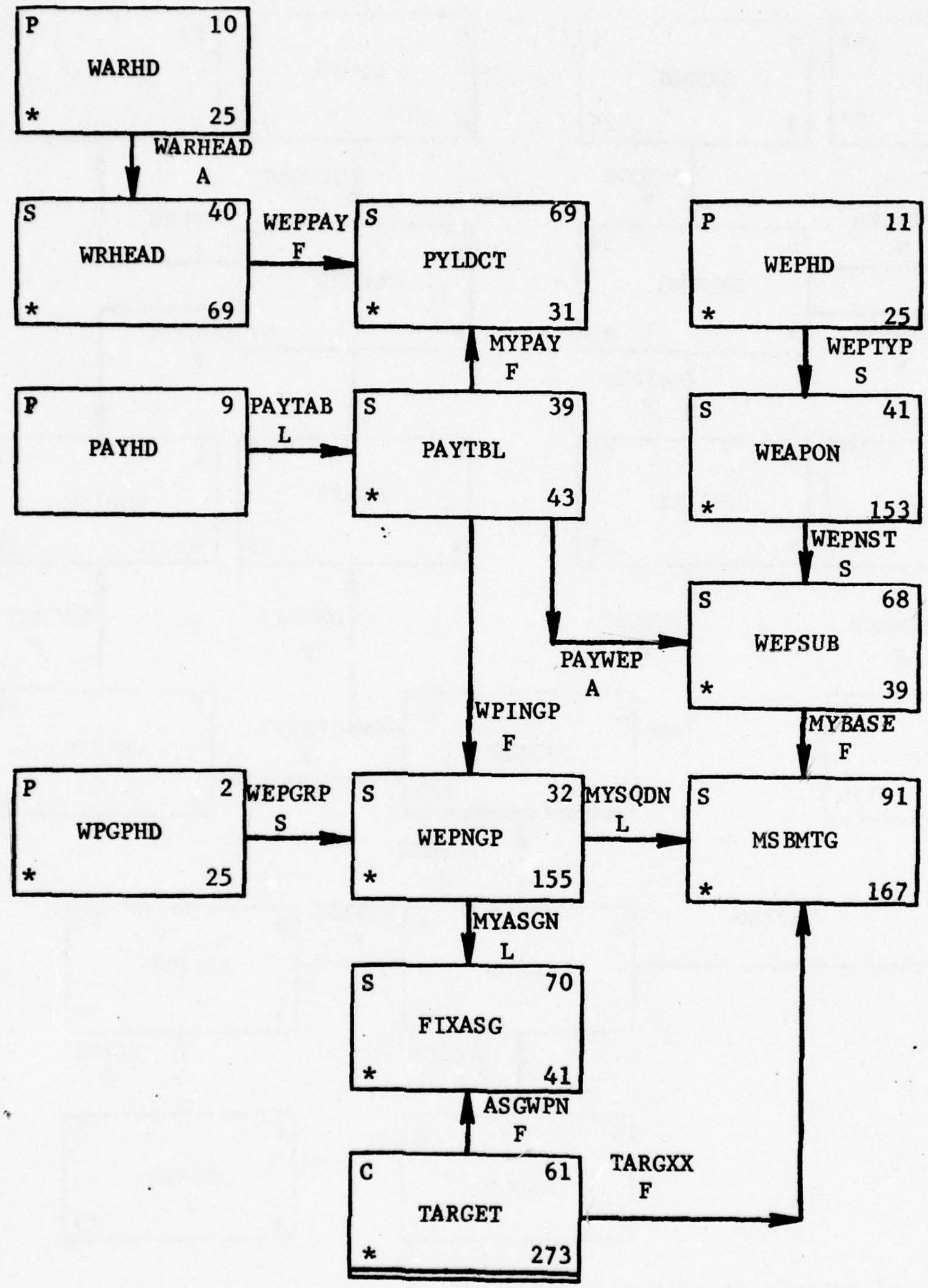

* Number of records is scenario dependent.

Figure 6. Scenario Data Structure



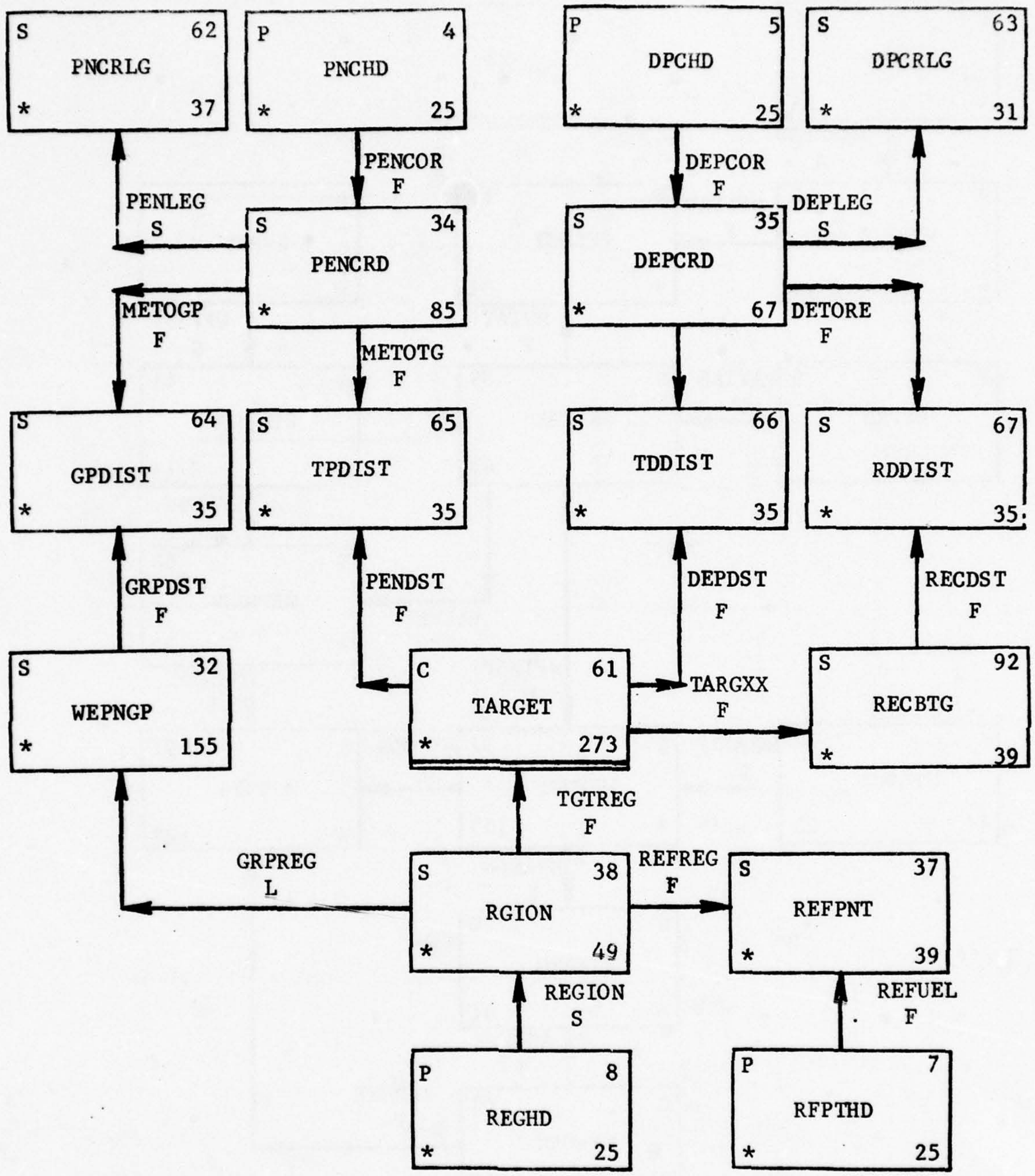

* Number of records is scenario dependent.

Figure 7. Target Data Structure




* Number of records is scenario dependent.

Figure 8. Weapon Data Organization




* Number of records is scenario dependent.

Figure 9. Geographic Data Structure

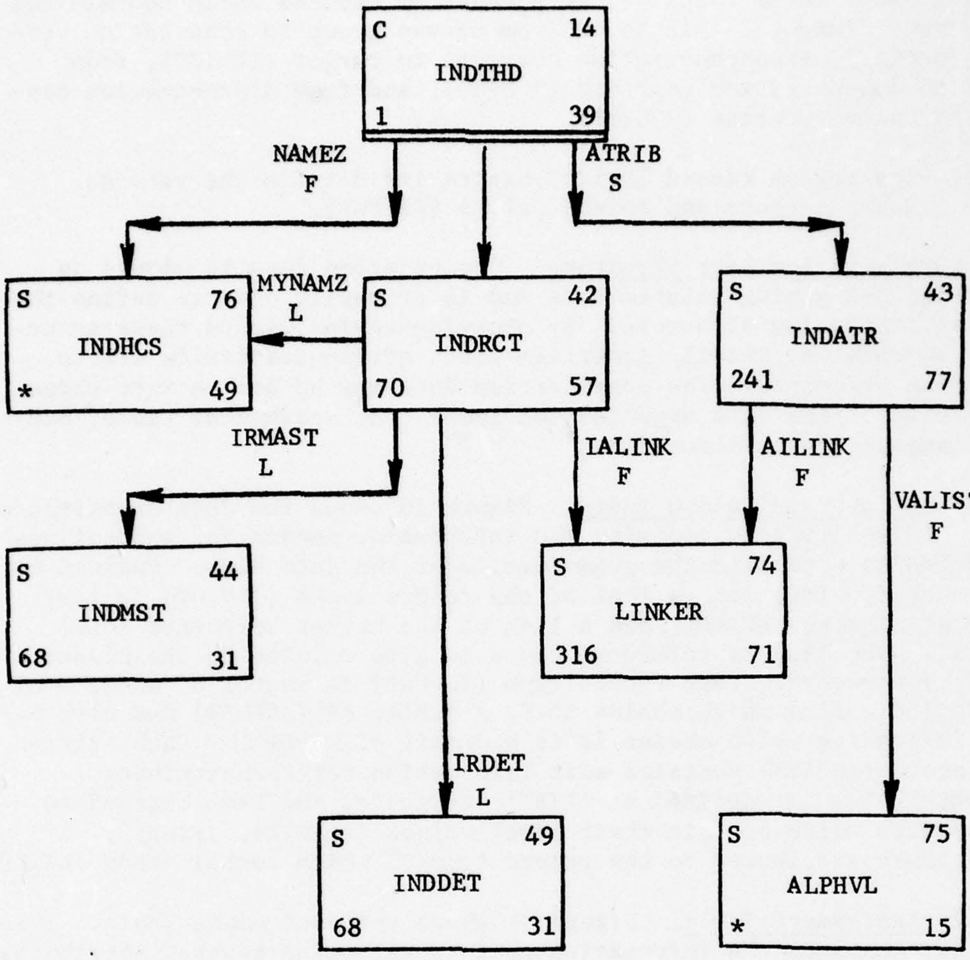
weapon groups, targets and recovery bases by records which contain the represented distance. Distances from weapon group to penetration corridor (GPDIST), from penetration corridor to target (TPDIST), from target to depenetration corridor (TDDIST), and from depenetration corridor to recovery bases (RDDIST).

Finally, the region record (RGION) has as its details the records: weapon groups, targets and refuel points (REFPNT).

2.4.2 Organization Data Structure. Organization data is viewed as containing non-gaming related data and is primarily used to define the shape of the gaming structure. By querying chains within these structures, modules may readily ascertain where attributes reside within the gaming structure. The organization data may be broken into three subdivisions; the data organization index, the assignment table, and miscellaneous organization data.

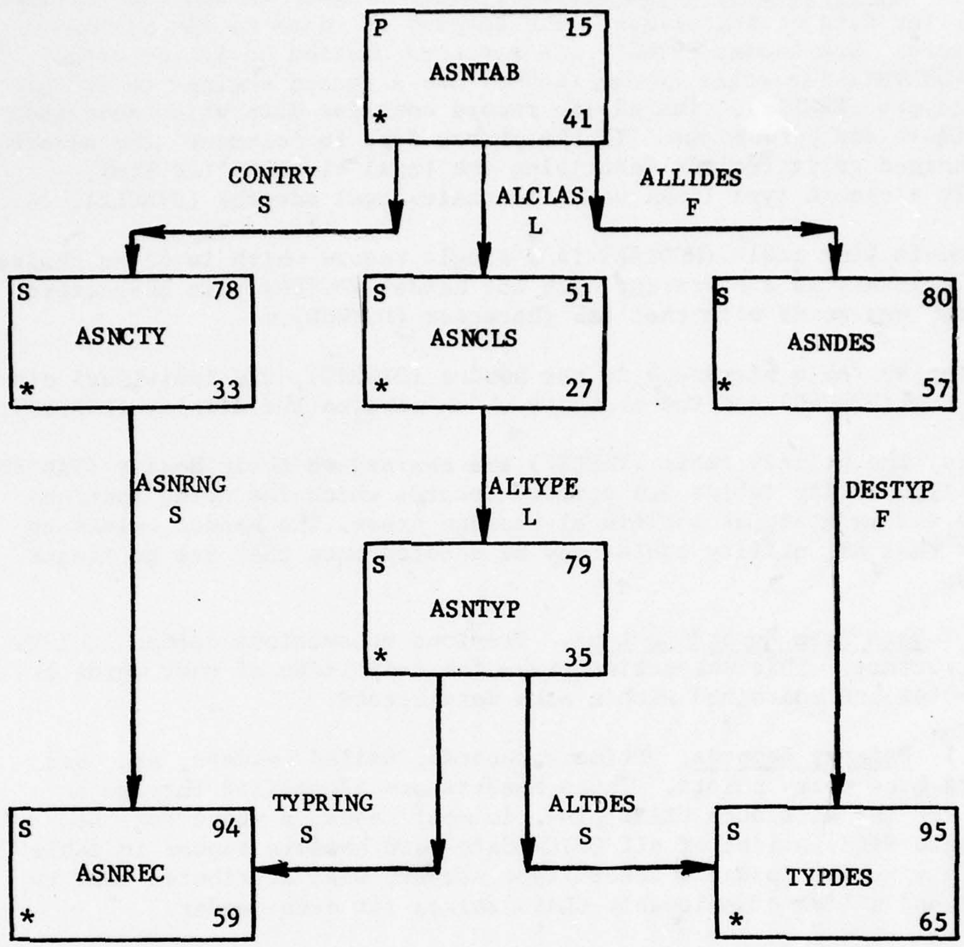
2.4.2.1 Data Organization Index. Figure 10 shows the data organization index. This hierarchy contains the information needed for subroutines or modules to determine the organization of the data base. Chained to the header (INDTHD) is: a list of the record types (INDRCT), a list of the attributes (INDATR), and a list of the header reference codes (INDHCS). The list of reference codes is also related to the header record type record. Each record type (INDRCT) is master of a chain of records indicating which chains it is a master of (INDMST) and also a chain indicating which chains it is a detail of (INDDET). Each attribute record (INDATR) contains edit information for the attribute. Some attributes are defined as 'LIST' attributes and have chained to them records which contain their legal values (ALPHVL). Finally, a record links attributes to the record type(s) which contain them (LINKER).

2.4.2.2 Assignment Table. Figure 11 shows the Assignment Table. This hierarchy contains the information used to determine several attributes for targets based on information from a JAD format record. Chained to the header (ASNTAB) for each side are records containing the valid country codes and the region they are in (ASNCTY). These records are sorted on region and country code. Also under each header, are records (ASNCLS) containing the values for CLASS for the side. Under each of these records are all the TYPE names that belong to this class (ASNTYP). The countries and types are connected via common ASNREC records. These records contain restrictions based on Category Code, the location or owner of the target, and either its name or size. It also contains the TASK that will be assigned to a target meeting these restrictions. On the other chain under TYPE are the alphabetic portions of DESIG to be used in assigning a DESIG to the target (TYPDES). Subsequent DESIGs are used if the first values are already used. All of these records with the same alphabetic portion of DESIG are chained together under a common record (ASNDES) containing the DESIG and the number in each region.




* Number of Records is scenario dependent.

Figure 10. Data Organization Index




* Number of records is scenario dependent.

Figure 11. Assignment Table

2.4.2.3 Miscellaneous Organizational Data. Figure 12 shows a number of smaller data organizations. The largest of these is the syntax directory. One header (SYNHD) has a record chained to it for each verb (SYNVBB). The other header (ADVHD) has a record chained to it for each adverb (PRMADV). The adverb record contains data which describes the clause and phrase type. If the phrase type is 'element' the adverb has chained to it records describing the legal elements (ADVELM). Finally a record type links verbs to their legal adverbs (SYNCLZ).

The module link table (MODTAB) is a single record which is on no chains. The dictionary is a hierarchy with the header (DCTHD), tab characters (DCTTAB) and words with that tab character (DCTWRD).

The display table hierarchy is the header (DISPHD), the individual display name (DISPRC) and the elements which make up the display (DISPDT).

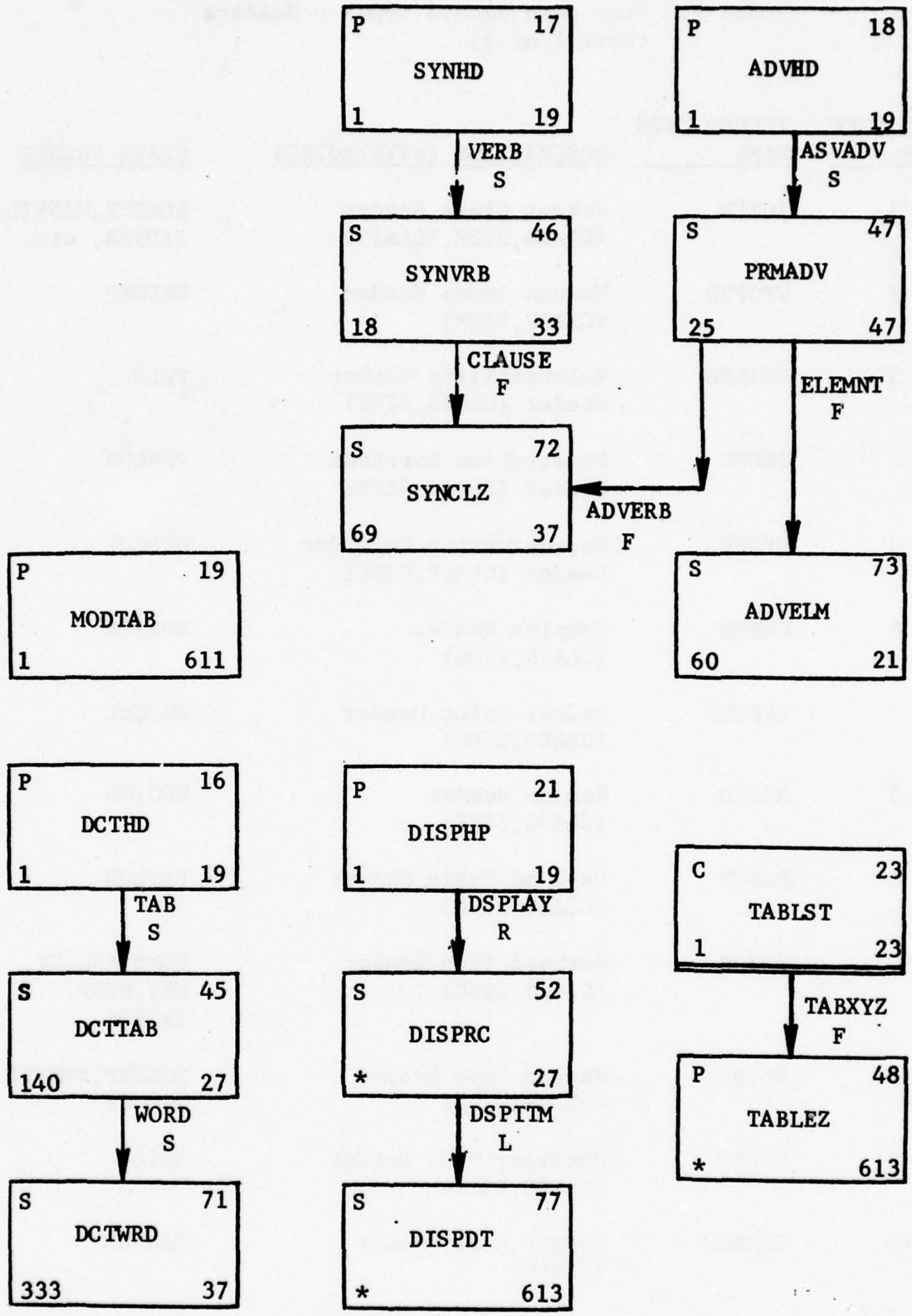
Finally, the utility table (TABLEZ) are chained to their header (TABLST). Actually, utility tables are primary records which the using routines create and maintain as additional storage areas. The header exists to assure that all utility tables may be deleted once they are no longer needed.

2.4.3 Data Base Record Content. Previous subsections defined QUICK's IDS structure. This subsection gives the definition of what words or attributes are contained within each data record.

2.4.3.1 Primary Records. Primary records, called headers, are used as data base entry points. These headers are identified through a value for the attribute CLASS plus, in most cases, a value for the attribute SIDE. A list of all QUICK data base headers appear in table 9 along with the header's record type number, what attributes need to be set and a list of allowable CLASS values for each header.

2.4.3.2 Secondary Records. The remainder of the record types are secondary records (the TARGET record is a CALC record). Each secondary record is a collection of attributes and/or internally used values (table 10).

A list of QUICK data base chains appears in table 11. All chains in the data base structure are 'linked to prior' so that when any record type is deleted, the physical file space it occupies is released for other use. Many of the chains are also 'linked to master' to speed processing when a subroutine calls entry point HEAD for those chains (see table 12).




* Number of records is scenario dependent.

Figure 12. Miscellaneous Organizational Data

Table 9. Data Base Record Types - Headers (Part 1 of 2)

| RECORD TYPE NUMBER | RECORD TYPE NAME | DESCRIPTION (ATTRIBUTES) | CLASS VALUES |
|---|---|---|---|
| 1 | TGTHD | Target Class Header (CLASS,SIDE,ICLASS) | BOMBER,MISSIL, TANKER, etc. |
| 2 | WPGPHD | Weapon Group Header (CLASS,SIDE) | WEPGRP |
| 3 | VNTKHD | Vulnerability Number Header (CLASS,SIDE) | VULN |
| 4 | PNCHD | Penetration Corridor Header (CLASS,SIDE) | PENCOR |
| 5 | DPCHD | Depenetration Corridor Header (CLASS,SIDE) | DEPCOR |
| 6 | CMPHD | Complex Header (CLASS,SIDE) | COMPLX |
| 7 | RFPTHD | Refuel Point Header (CLASS,SIDE) | REFUEL |
| 8 | REGHD | Region Header (CLASS,SIDE) | REGION |
| 9 | PAYHD | Payload Table Header (CLASS,SIDE) | PAYLOD |
| 10 | WARHD | Warhead Type Header (CLASS,SIDE) | BOMB,ASM,RV, MRV,MIRV, FACTOR |
| 11 | WEPHD | Weapon Type Header (CLASS,SIDE) | MSLWEP,BMBWEP TNKWEP |
| 12 | RCBHD | Recovery Base Header (CLASS,SIDE) | RECOV |
| 13 | TARNUM | Target List Header (CLASS) | TARNUM |

Table 9. (Part 2 of 2)

| RECORD TYPE NUMBER | RECORD TYPE NAME | DESCRIPTION (ATTRIBUTES) | CLASS VALUES |
|---|---|---|---|
| 14 | INDTHD | Data Organization Index Header (CLASS) (This is a CALC record) | INDEX |
| 15 | ASNTAB | Assignment Table Header (CLASS, SIDE) | ASSIGN |
| 16 | DCTHD | Dictionary Header (CLASS) | DICTON |
| 17 | SYNHD | Syntax Directory Verb Header (CLASS) | SYNTAX |
| 18 | ADVHD | Syntax Directory Adverb Header (CLASS) | ADVERB |
| 19 | MODTAB | Module Link Table (CLASS, link Table (100 words)) | MODTAB |
| 20 | SECHD | Sector Header (CLASS) | SECTOR |
| 21 | DISPHD | REPORT Module Display Header (CLASS) | DIPLY |
| 22 | NUMTBL | General Number Table (CLASS) | NUMBER |
| 23 | TABLST | Utility Table Header (CLASS) (This is a CALC record) | TABLST |

Table 10. Data Base Record Types - Secondary Records (Part 1 of 4)

| RECORD NUMBER | TYPE NAME | DESCRIPTION (ATTRIBUTES) |
|---|---|---|
| 31 | TARGTY | Target Type (CNTRYLO, CNTRYOW, FLAG, FVALT1, FVALT2, FVALT3, FVALT4, FVALT5, FVULN1, NHRDCOMP, NTIMCOMP, T1, T2, T3, T4, T5, TYPE, VULN1 VULN2) |
| 32 | WEPNGP | Weapon Group (GBASE, GFRASM, GLAT, GLONG, GNWPNS, GNVEH, GPKNAV, GREFCODE, GREFTIME, GROUP, GSBL, GSBLREAL, GSTART, GTYPE, GYIELD, IALERT, IREG) (GROUP is used as a Match Key) |
| 33 | VULNUM | Vulnerability Number (VULN1) (VULN1 is a match key) |
| 34 | PENCRD | Penetration Corridor (ATTRCO, ATTRSU, CORNUM, DEFRAN, HILOAT, KORSTY, ORLAT, ORLONG) |
| 35 | DEPCRD | Depenetration Corridor (CORNUM, MYRECOV1, MYRECOV2, MYRECOV3, MYRECOV4) |
| 36 | COMPTG | Complex Target (CNTRYLO, CNTRYOW, DESIG, FLAG, FVALT1, FVALT2, FVALT3, FVALT4, FVALT5, FVULN1, HAZ2, HGZ, GHZ2, ICOMPL, IDHOB, INDEXNO, LAT, LONG, MAXFRA, MAXKILL, MINKILL, MISDEF, NAME, NHRDCOMP, NTIMCOMP, NTINT, RADIUS, TARDEFHI, TARDEFLO, TASK, TGTMULT, T1, T2, T3, T4, T5, VALUE, VOZ) (ICOMPL is a match-key) |
| 37 | REFPNT | Refuel Point (IREG, LAT, LONG) |
| 38 | RGION | Region (IREG, CCREL) (IREG is a match key) |
| 39 | PAYTBL | Payload table (PAYTBLNM) |
| 40 | WRHEAD | Warhead (CEPASM, FFRAC, NAREADEC, NCMS, NDECOYS, NWADS, PAYALT, PDUD, RANGEASM, RELASM, SPEEDASM, TYPE, YIELD) |
| 41 | WEAPON | Weapon type (ACTIVE, ALTDLY, CEP, CMISS, FUNCTI, IPENMO, IRECMO, IREP, LCHINT, MAXSAL, NALTDLY, NMPSIT, PDES, PFPF, PINC, PLABT, RRABT, RANGE, RANGEDEC, RANGEREF, REL, RNGMIN, SIMLUN, SPDLO, SPEED, TOFMIN, TTOS, TYPE) (TYPE is a match key) |
| 42 | INDRCT | Index Record Type Record (contains record type name and number) |

Table 10. (Part 2 of 4)

| RECORD NUMBER | TYPE NAME | DESCRIPTION (ATTRIBUTES) |
|---|---|---|
| 43 | INDATR | Index Attribute Record (ATTRIBN1, ATTRIBN2, ATTRIBNO, ATTRBTYP, ATTRIBAD, ATDEFALT, ATTRNGHI, ATTRNGLO) (ATTRIBN1 and ATTRIBN2 are match keys) |
| 44 | INDMST | Index Master Record (CHAINNAM, MASDETNM, MASDETNO) |
| 45 | DCTTAB | Dictionary Tab-character (TABCHAR) (TABCHAR is a match key) |
| 46 | SYNVRB | Syntax Verb (CLAUSESW, VERBVAL) (VERBVAL is a matcky key) |
| 47 | PRMADV | Syntax Adverb (ADVERBVL, CLAUSETY, PHRASETY) (ADVERBVL is a match key) |
| 48 | TABLEZ | Utility Table (contains 100 words which are module defined) |
| 49 | INDDET | Index Detail Record (CHAINNAM, MASDETNM, MASDETNO) |
| 50 | SECTR | Sector (LONGLOWR, LONGUPER) |
| 51 | ASNCLS | Assignment Table Class (ACLASS) |
| 52 | DISPRC | Display Record (DISPNAM1, DISPNAM2) |
| 61 | TARGET | Target (BENO, CATCODE, DESIG, HAZ, HAZ2, HGZ, HGZ2, ICOMPL, IDHOB, IGIW, INDEXNO, IREG, ISITE, LAT, LONG, MAJOR, MAXFRA, MAXKILL, MINKILL, MINOR, MISDEF, NAME, NTINT, POP, RADIUS, SIDE, TARDEFHI, TARDEFLO, TASK, VALUE, VOZ, WACNO) (Record is a CALC record - randomized on DEGIG) |
| 62 | PNCRLG | Penetration Corridor Leg (ATTRLE, DOGLEG, LAT, LONG) |
| 63 | DPCRLG | Depenetration Corridor Leg (DOGLEG, LAT, LONG) |
| 64 | GPDIST | Distance from Group to Penetration Corridor (DISTANCE) |
| 65 | TPDIST | Distance from Target to Penetration Corridor (DISTANCE) |

Table 10. (Part 3 of 4)

| RECORD NUMBER | TYPE NAME | DESCRIPTION (ATTRIBUTES) |
|---|---|---|
| 66 | TDDIST | Distance from Target to Depenetration Corridor (DISTANCE) |
| 67 | RDDIST | Distance from Recovery Base to Depenetration Corridor |
| 68 | WEPSUB | Weapon Subtype (WEPNAME) (WEPNAME is a match key) |
| 69 | PYLDCT | Count of warhead type in Payload Table (NUMLOAD) |
| 70 | FIXASG | Fixed assignment of Weapon Group to Target (ARRIVE, NUMASSGN) |
| 71 | DCTWRD | Dictionary Word (WORDSTR1, WORDSTR2, WORDTY, WORDVL) |
| 72 | SYNCLZ | Links Verbs to Adverbs (ADVERBVL, VERBVAL) |
| 73 | ADVELM | Gives Legal Elements for elemental Adverbs (ELEMNTTY, ELEMNTVL) |
| 74 | LWKER | Links Attributes to Record Types (ATTRIBN1, ATTRIBN2, ATTRIBAD, ATTRBTYP, contains 2 wrods which are not attributes) |
| 75 | ALPHVL | Legal Values for 'LIST' Attributes (ALPLSTVL) |
| 76 | INDHCS | Stores Header reference codes (contains header reference code (two words), and the associated CLASS and SIZE values) |
| 77 | DISPDT | Display table list (contains 100 words which are created by REPORT) |
| 78 | ASNCTY | Assignment Table County (COUNTRY, REGION) |
| 79 | ASNTYP | Assignment Table Type (ATYPE) |
| 80 | ASNDES | Assignment Table Desig (DESIGA2, KOUNT1, KOUNT2, KOUNT3, KOUNT4, KOUNT5) |
| 91 | MSBMTG | Missile Bomber Target (ADBLI, ADBLR, ALRTDB, ALRTDL, GROUP, IREFUEL, NADBLI, NADBLR, NLRTDB, NLRTDL, NOALER, NOINCO, NOPERSQ, NPRSQ1, NPRSQ2, NPRSQ3, NPRSQ4, NUMDBL, PKNAV, VONBASE, WEPNAME) |

Table 10. (Part 4 of 4)

| RECORD NUMBER | TYPE NAME | DESCRIPTION (ATTRIBUTES) |
|---|---|---|
| 92 | RECBIG | Recovery base (CAPACITY) |
| 93 | TARCDE | Target List (TGTNUMB, TGTREFCD) |
| 94 | ASNREC | Assignment Table Category (ASNTASK, CATHI, CATLO, CNFLG, MWCAP) |
| 95 | TYPDES | Assignment Table Type DESIG (DESIGA2, FULL1, FULL2, FULL3, FULL4, FULL5) |

Table 11. Data Base Chains (Part 1 of 4)

| CHAIN NAME | MASTER RECORD | DETAIL RECORD | DESCRIPTION |
|---|---|---|---|
| ADVADV | ADVHD | PRMADV | Links Adverb Header to adverbs |
| ADVERB | PRMADV | SYNCLZ | Links adverb to link to verb |
| AILINK | INDATR | LINKER | Links Attribute Record to link to record type |
| ALCLAS | ASNTAB | ASNCLS | Links Assignment table header to assigned classes |
| ALLDES | ASNTAB | ASNDES | Links Assignment table header to assigned DESIG |
| ALTDES | ASNTYP | TYPDES | Links Assigned type to link to DESIGs for that type |
| ALTYPE | ASNCLS | ASNTYP | Links Assigned class to assigned types in that class |
| ASGWPN | TARGET | FIXASG | Links target to fix assignments to it |
| ASNRNG | ASNCTY | ASNREC | Links Assigned country to category ranges for that country |
| ATRIB | INDTHD | INDATR | Links index header to attribute record |
| CLAUSE | SYNVRB | SYNCLZ | Links verb to link to adverbs |
| CMPTGT | COMPTG | TARGET | Links complex to targets which make up the complex |
| COMPLX | CMPHD | COMPTG | Links complex header to complexes |
| CONTRY | ASNTAB | ASNCTY | Links assignment table header to assigned countries |
| DEPCOR | DPCHD | DEPCRD | Links depenetration corridor header to depenetration corridors |
| DEPDST | TARGET | TDDIST | Links target to distance to depenetration corridors |
| DEPLEG | DEPCRD | DPCRLG | Links depenetration corridors to its doglets |
| DESTYP | ASNDES | TYPDES | Links assigned DESIG to link to assigned type |
| DETORE | DEPCRD | RDDIST | Links depenetration corridor to distance to recovery base |

Table 11. (Part 2 of 4)

| CHAIN NAME | MASTER RECORD | DETAIL RECORD | DESCRIPTION |
|---|---|---|---|
| DETOTG | DEPCRD | TDDIST | Links depenetration corridor to distance to target |
| DSPITM | DISPRC | DISPDT | Links display table to its elements |
| DSPLAY | DISPHD | DISPRC | Links display header to display tables |
| ELEMNT | PRMADV | ADVELM | Links elemental adverb to its legal elements |
| GRPDST | WEPNGP | GPDIST | Links weapon group to distance to penetration corridor |
| GRPREG | RGION | WEPNGP | Links region to weapon groups in that region |
| IALINK | INDRCT | LINKER | Links record type to link to attributes |
| IRDET | INDRCT | INDDET | Links record type to record showing chains of which it is a detail |
| IRMAST | INDRCT | INDMST | Links record type to record showing chains of which it is master |
| LISTXX | TARNUM | TARCDE | Links target list header to elements of the list |
| METOGP | PENCRD | GPDIST | Links penetration corridor to distance to weapon group |
| METOTG | PENCRD | TPDIST | Links penetration corridor to distance to target |
| MYASGN | WEPNGP | FIXASG | Links weapon group to fixed assignments |
| MYBASE | WEPSUB | MSBMTG | Links weapon sub-type to missile/bomber targets that are its bases |
| MYNAMZ | INDRCT | INDHCS | Links record type for headers to their reference codes |
| MYPAY | PAYTBL | PYLDCT | Links payload table to warhead type count |
| MYSQDN | WEPNGP | MSBMTG | Links weapon group to missile/bomber targets which provide bases for the group |
| NAMEZ | INDTHD | INDHCS | Links index header to header reference codes |

Table 11. (Part 3 of 4)

| CHAIN NAME | MASTER RECORD | DETAIL RECORD | DESCRIPTION |
|---|---|---|---|
| PAYTAB | PAYHD | PAYTBL | Links payload table header to payload tables |
| PAYWEP | PAYTBL | WEPSUB | Links payload table to weapon sub-type that uses it |
| PENCOR | PNCHD | PENCRD | Links penetration corridor header to penetration corridors |
| PENDST | TARGET | TPDIST | Links target to distance to penetration corridor |
| PENLEG | PENCRD | PNCRLG | Links penetration corridor to its doglegs |
| RCTYP | INDTHD | INDRCT | Links index header to record types |
| RECDST | RECBTG | RDDIST | Links recovery base to distance to depentration corridors |
| RECOVB | RCBHD | RECBTG | Links recovery base header to recovery bases |
| REFREG | RGION | REFPNT | Links region to refuel points in the region |
| REFUEL | RFPTHD | REFPNT | Links refuel point header to refuel points |
| REGION | REGHD | RGION | Links region header to regions |
| SECTOR | SECHD | SECTR | Links sector header to sectors |
| TAB | DCTHD | DCTTAB | Links dictionary header to its tab characters |
| TABXYZ | TABLST | TABLEZ | Links utility table header to utility tables |
| TARGXX | TARGET | MSBMTG | Links target to missile/bomber target additional data |
| TARGXX | TARGET | RECBTG | Links target to recovery base additional data |
| TGTREG | RGION | TARGET | Links region to targets in region |
| TGTSEC | SECTR | TARGET | Links sector to targets in sector |
| TGTTGT | TARGTY | TARGET | Links target type to targets of that type |

Table 11. (Part 4 of 4)

| CHAIN NAME | MASTER RECORD | DETAIL RECORD | DESCRIPTION |
|---|---|---|---|
| TGTTYP | TGTHD | TARGTY | Links target header to target types |
| TGTVNS | VULNUM | TARGTY | Links vulnerability number to target types with that number |
| TYPRNG | ASNTYP | ASNREC | Links assigned type to category range for that type |
| VALIST | INDATR | ALPHVL | Links 'list' attribute to its legal values |
| VERB | SYNHD | SYNVRB | Links syntax header to verbs |
| VNTKS | VNTKHD | VULNUM | Links vulnerability number header to vulnerability number |
| WARHED | WARHD | WRHEAD | Links warhead header to warhead types |
| WEPGRP | WPGPHD | WEPNGP | Links weapon group header to weapon groups |
| WEPNST | WEAPON | WEPSUB | Links weapon type to weapon subtype |
| WEPPAY | WRHEAD | PYLDCT | Links warhead type to count in payload table |
| WEPTYP | WEPHD | WEAPON | Links weapon header to weapon types |
| WORD | DCTTAB | DCTWRD | Links tab character to words with that tab |
| WPINGP | PAYTBL | WEPNGP | Links payload table to weapon groups using that table |

Table 12. Chains Which are Linked to Master

| CHAIN NAME | MASTER RECORD | DETAIL RECORD |
|---|---|---|
| ADVERB | PRMADV | SYNCLZ |
| AILINK | INDATR | LWKER |
| ALTDES | ASNTYP | TYPDES |
| ASGWPN | TARGET | FIXASG |
| ASNRNG | ASNCTY | ASNREC |
| CMPTGT | COMPTG | TARGET |
| DEPDST | TARGET | TDDIST |
| DETORE | DEPCRD | RDDIST |
| DETOTG | DEPCRD | TDDIST |
| DESTYP | ASNDES | TYPDES |
| GRPDST | WEPNGP | GPDIST |
| IALWK | INDRCT | LINKER |
| METOGP | PENCRD | GPDIST |
| METOTG | PENCRD | TPDIST |
| MYASGN | WEPNGP | FIXASG |
| MYBASE | WEPSUB | MSBMTG |
| MYNAMZ | INDRCT | INDHCS |
| MYSQPN | WEPNGP | MSBMTG |
| PAYWEP | PAYTBL | WEPSUB |
| PENDST | TARGET | TPDIST |
| RECDST | RECBTG | RDDIST |
| TARGXX | TARGET | MSBMTG |
| TARGXX | TARGET | RECBTG |
| WEPPAY | WRHEAD | PYLDCT |

## 3. CENTRAL OPERATIONS PROCESSOR

### 3.1 Purpose

The Central Operations Processor (COP), or QUICK's executive software module, acts as an intermediary between the user and the operating system, and provides the communications between the QUICK system and the integrated data base. The COP acts accordingly by interpreting user commands written in imperative text English formats.

The major subroutines of the COP provide the following functions:

- o Data base interface
- o Text English syntax analysis
- o Input text English translation
- o Module execution
- o Organizational data initialization

### 3.2 Input

The COP operates in two modes insofar as input is concerned. The BOOT subroutine requires no previous entries to have been made to the data base. Indeed the BOOT subroutine's function is to initialize the organizational data required for normal COP runs. The organizational data may also be provided by the use of the RESTORE verb to read in a previously created QUICK save tape (see section 7). For a normal COP run it is required that the organizational data have been initialized.

### 3.3 Output

The output of a normal COP run is the array which contains the instruction codes resulting from the translation of the input text English. If the BOOT subroutine is used at the beginning of the run, the organizational data may be initialized or updated.

### 3.4 Concept of Operation

The COP first calls upon its INICOP subroutine to initialize headers and/or checks to see if BOOT or RESTORE are being called before normal processing. The normal operating mode is to process each input sentence separately in three steps.

First, subroutine ERRFND is called to read in the sentence and check its syntax. The text English input is read one character string at a time. As each string is read, it is examined to determine what type of language element it is. Some strings are obviously operators or numerics, all others are 'looked up' in the dictionary. Once the type is determined,

the string is checked against those preceding it for proper syntax. If the syntax checks, the string is converted into a symbol meaningful to the input translator and stored in a table (see section 3.5.2 below). This procedure continues until a syntax error is discovered or a verb is read (the first character string should, of course, be a verb but processing does not halt until the second verb).

The second step in normal COP processing is to call subroutine INPTRN to translate the input. INPTRN uses the table of symbols created by ERRFND to form a sequence of instruction codes. These instruction codes are stored in the Instruction Code Array which is maintained by INSPUT.

The final step is to call subroutine MODGET which uses the input sentences verb to find the desired overlay link name in the module link table. The overlay is read in and the module is executed.

If at any time during processing the error flag (common block OOPS) is set on, the INPTRN and MODGET routines are bypassed. ERRFND is still called to analyze the syntax of remaining command sentences.

In the case of the BOCT subroutine, the input card images are read directly by the BOOT subroutine. Each card image, in general, calls for creation of or modification to a particular record type in the organizational data. Once the BOOT input is complete, BOOT returns control to COP to read the next verb

### 3.5 Identification of Subroutine Functions

3.5.1 Data Base Interface. This function is performed by subroutines QDATA, HDFND, and INSPUT. Their use is fully discussed in section 2.

3.5.2 Text English Syntax Analysis. This function is performed by the ERRFND subroutine. The process is to call GETSTR for the next input string. If the string is a 'long string' LNGSTR is called to save it in the symbol tables. If the string is an alphabetic, WEBSTR is called to look-up the string in the dictionary. Next, SYNTAX is called to check the syntax of the sentence. The TABINS routine is used to build a set of four tables. The tables are maintained in the TABLZ common block. If more space is needed for any of the four tables, a TABLEZ record is stored and its reference code retained in the TABLZ common block. This group of tables is also used by the TABGET, and LINEIO subroutines of INPTRN. The four tables maintained by TABINS contain:

- o A list of all unique numeric constants in the input sentence
- o A list of all unique attributes in the input sentence
- o A list of all unique alphabetic constants in the input sentence
- o A list of symbols which represent the input string

Table 13. ERRFND Table Symbols

| INDICATOR (BITS 31-35) | REMAINDER (BITS 0-30) |
|---|---|
| 1 (Operator) | Operator identifying number |
| 2 (Long String) | Number of characters in long string. This symbol is followed by the number of alphabetic constant symbols (indicator=9) needed to retain the entire long string. |
| 3 (Verb) | Verb identifying number |
| 4 (Adverb) | Adverb identifying number. The SYNTAX subroutine modifies this field by packing numbers that indicate the clause and phrase type as follows:<br>Remainder field = Adverb identifying value +100*Clause Type + 1000*Phrase Type where<br>Clause Type = { 1 - Boolian, 2 - Sequence, 3 - Single, 4 - Null }<br>Phrase Type = { 1 - Relational Phrase, 2 - Restricted Phrase, 3 - Elemental } |
| 5 (Special Word) | Special word identifying number |
| 6 (Attribute) | Index to attribute table which contains attribute values. The attribute table entry contains a single packed word as follows:<br>bits 0-1 = { 0 - Alphabetic, 1 - Numeric }<br>bit 2 = { 0 - Non-identifier, 1 - Identifier }<br>bits 3-17 = Attribute identifying number<br>bits 18-35 = Attribute's common block address |
| 9 (Alphabetic) | Index to alphabetic constant table. The table entry consists of two words. |
| 10 (Numeric) | Index to numeric constant table. The table entry consists of one word in floating point format. |

The last table mentioned contains one or more symbols for each input string. The symbols describe the string for the input translator. A summary of these symbols appears in table 13. Definition of identifying numbers can be found in Users Manual, Volume I.

3.5.3 Input Translation. This function is performed by the INPTRN subroutine. The tables built by ERRFND are translated into instruction codes and stored in the input instruction code tables. The instruction codes are discussed in section 2.

3.5.4 Module Execution. This function is performed by the MODGET subroutine. This subroutine uses the verb to determine the proper overlay, causes the overlay to be read in, and calls the module entry point (ENTMOD).

3.5.5 Organizational Data Initialization. This function is performed by the BOOT subroutine. BOOT reads card images which direct it to either create or modify records in the organizational data. The portions of the data base accessed via BOOT are:

- The data organization index
- The syntax analysis directory
- The dictionary
- The module link table

BOOT also creates headers and sectors.

## 3.6 Common Blocks

The common blocks used internally by the COP executive module as opposed to those used generally by all modules appear in table 14.

Table 14. Internal COP Common Block

| BLOCK | ARRAY OR VARIABLE | DESCRIPTION |
|---|---|---|
| C25 | | Represents record type INDHCS. Each record is used to identify the BCD reference code of a header. |
| | XCLASS | Header's CLASS value |
| | XSIDE | Header's SIDE value |
| | XREFCD | Header's BCD reference code (this variable is type character *8) |
| C35 | LINKS(100) | Module Link Table |
| FIRST | INCOM | Logical switch which, when on, indicates a sentence is being analyzed |
| SYMBOL | KYMBOL | Used to pass constructed ERRFND symbol (see section 3.5.2). |
| TABLZ | | Contains buffers for utility tables used to store ERRFND table |
| | KTBVAL(100,4) | Buffer of 100 words for each table type |
| | TBRFCD(20,4) | Contains Reference Codes for tables |
| | NUMOT(4) | Number of utility tables created |
| | NUMCT(4) | Index numbers of table currently in buffer. |

## 3.7 Main Routine of COP

| | |
|---|---|
| PURPOSE: | Main program of executive module |
| ENTRY POINTS: | COP (for purpose of discussion) |
| FORMAL PARAMETERS: | None |
| COMMON BLOCKS: | C15, IPQT, OOPS |
| SUBROUTINES CALLED: | CLZIDS, DELTAB, ERRFND, INICOP, INPTRN, INSDEL, MODGET |
| CALLED BY: | HIS operating system |

Method:

First, several switches are set: CHCKOV to control a later call to DELTAB, ERROR to indicate no error has occurred yet, and ENSW to indicate no end of input. Next, INICOP is called. The process which follows occurs for each command sentence.

First the ERRFND overlays are read in (this does not occur for the first sentence since INICOP has already done this). Next, ERRFND is called. If CHCKOV is still true, the INPTRN link is read in and INPTRN is executed if no error has occurred. If an error occurred before INPTRN, only DELTAB is called.

Next, if no error has occurred, MODGET is called to execute the module. If an error has occurred, CHCKOV is set to false.

Finally, the End Input switch is checked (ENDSW); if not on, the next input sentence is processed. If it is on, CLEZIDS is called and processing stops.

COP is illustrated in figure 13.

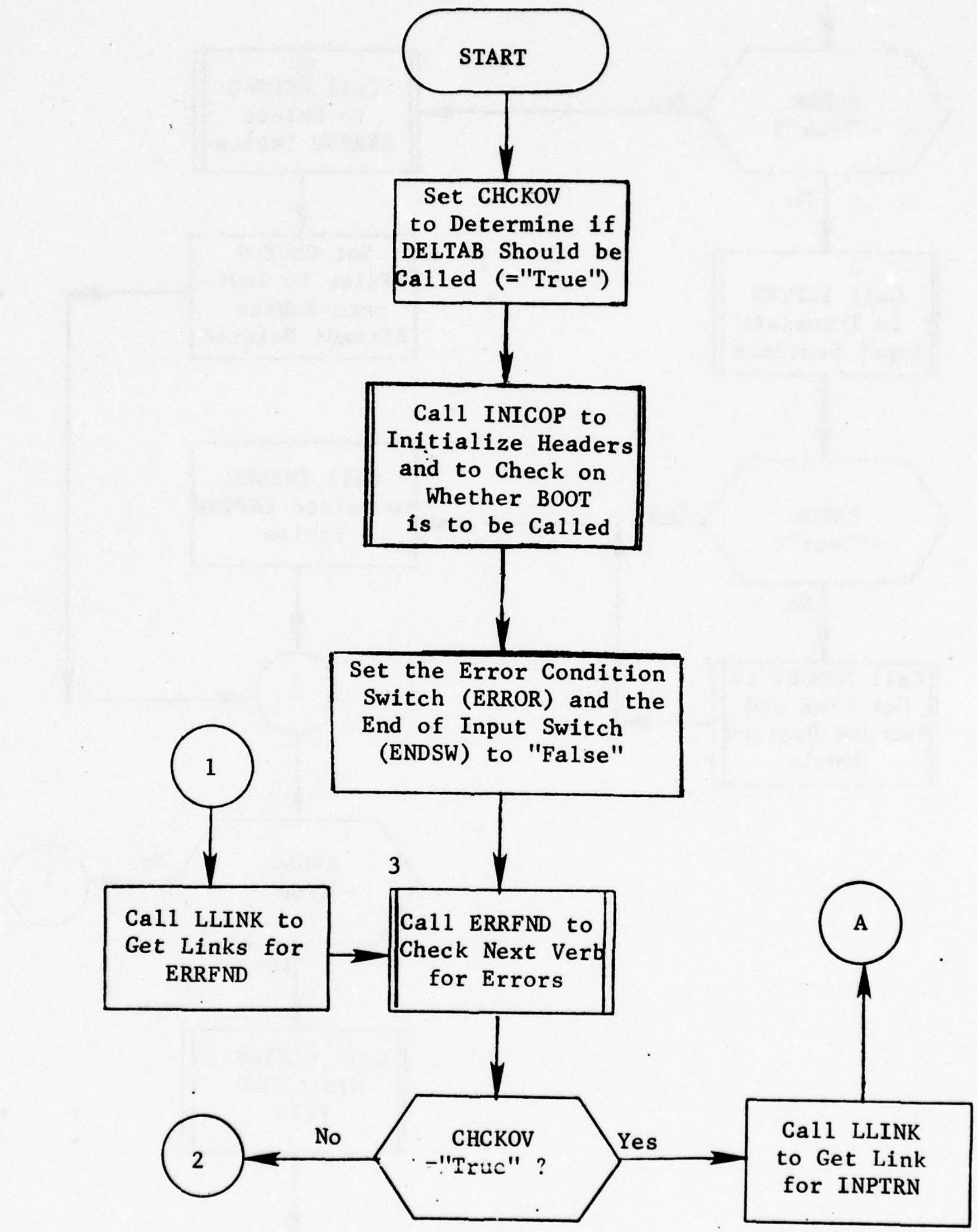



Figure 13. Program COP (Part 1 of 2)

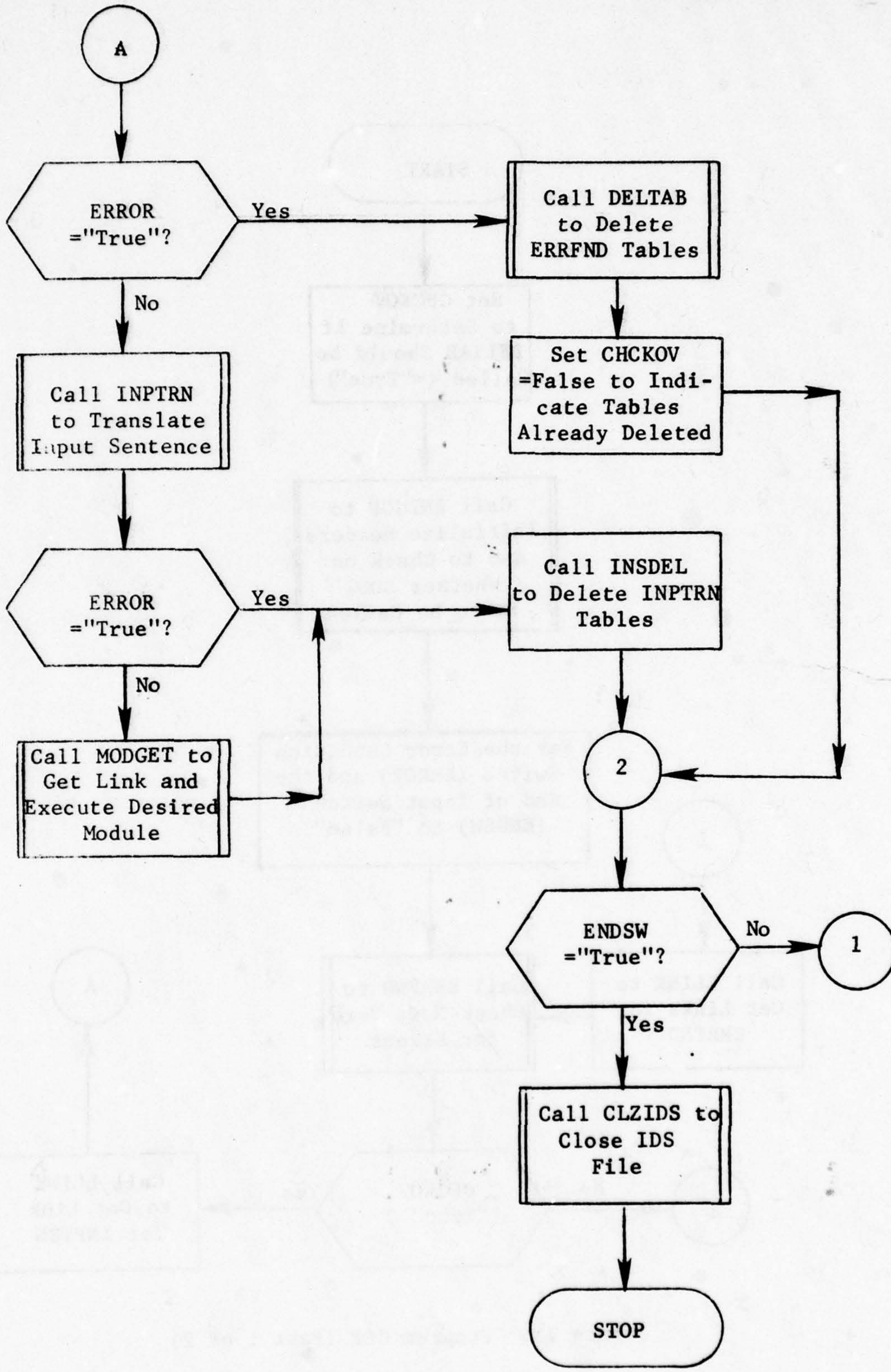



Figure 13. (Part 2 of 2)

### 3.7.1 Subroutine BANNER

PURPOSE: To produce banner print of module link name

ENTRY POINTS: BANNER

FORMAL PARAMETERS: PRINPT: Character string for banner

COMMON BLOCKS: IDAB

SUBROUTINES CALLED: None

CALLED BY: INICOP, MODGET

Method:

First, each character of PRINPT is extracted and stored in INPCHR. Also a number from 1-26 corresponding to the character's position in the alphabet is stored in INPTYP. If a character is not in the alphabet, zero is stored in INPTYP. If character five is blank, the characters are shifted one to the right. The output page is spaced so that the banner will be centered.

Now the following process is performed for 14 lines. For each character, a bit mask is retrieved from the IDABET array. This mask informs BANNER which character position to fill and which to leave blank. Once this function is performed for each of the six characters, the line is printed. When all lines are printed, BANNER spaces to the end of the page.

Subroutine BANNER is illustrated in figure 14.

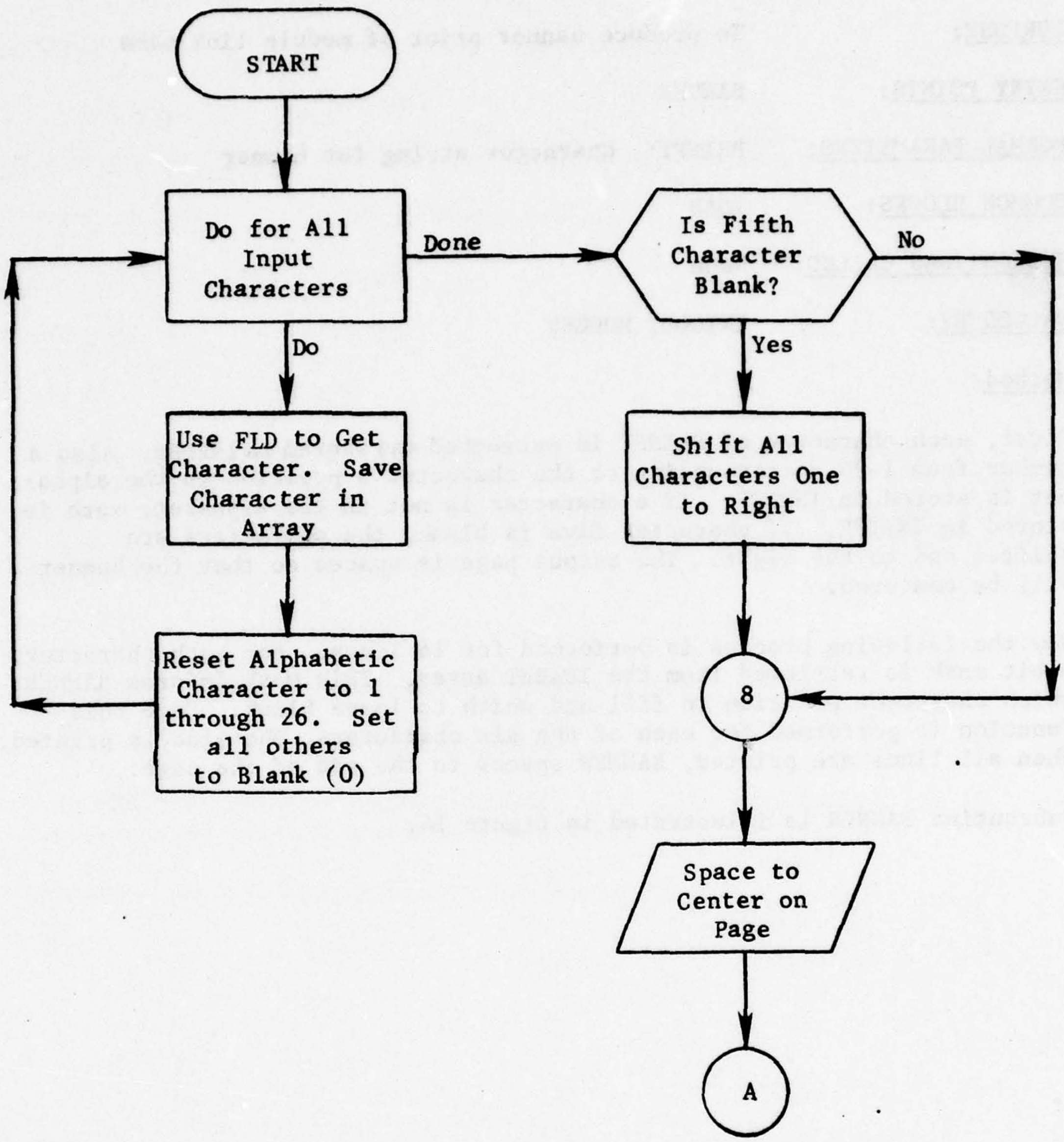



Figure 14. Subroutine Banner (Part 1 of 2)

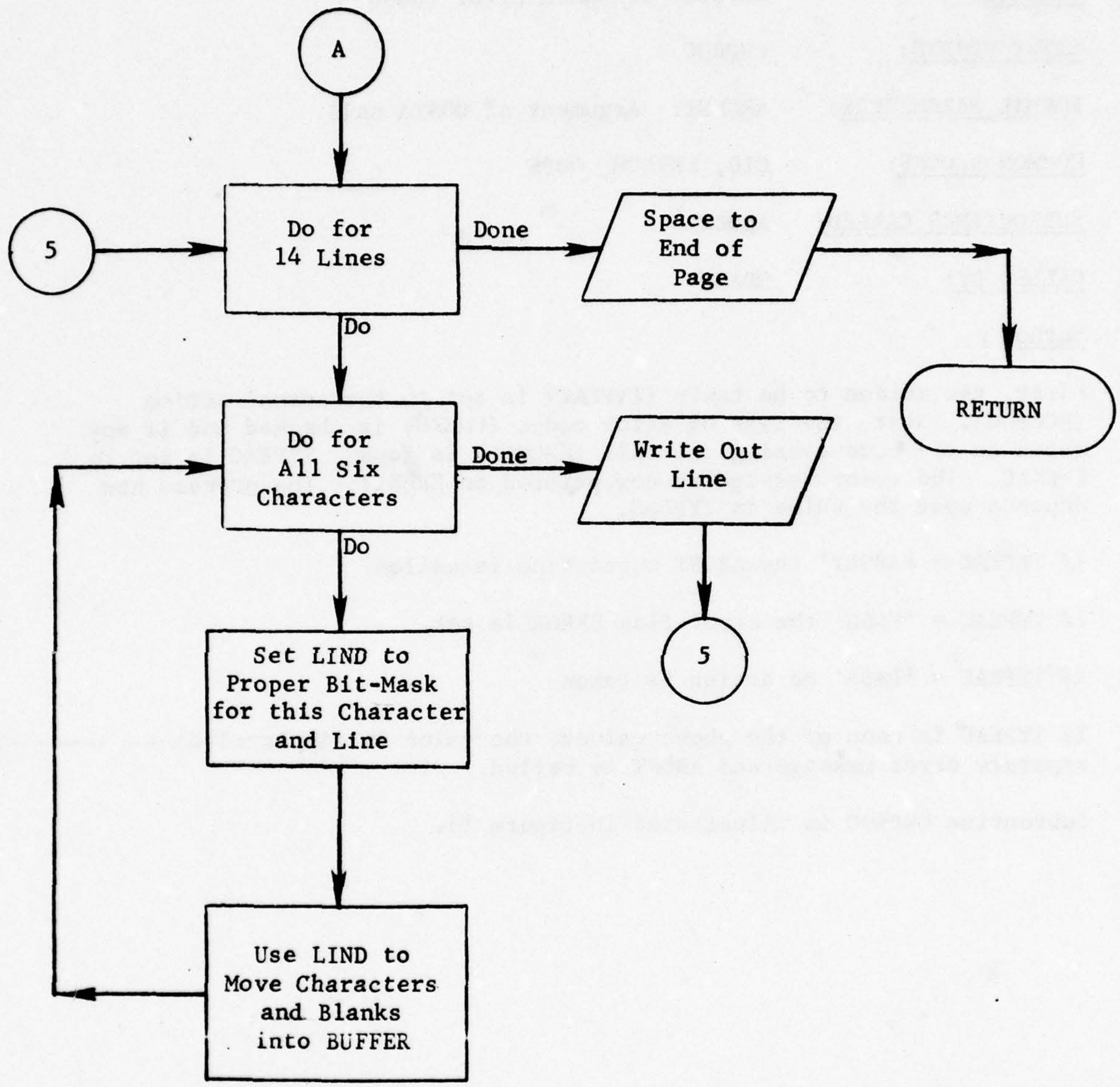



Figure 14. (Part 2 of 2)

### 3.7.2 Subroutine ERPROC

| | |
|---|---|
| PURPOSE: | To process QDATA error codes |
| ENTRY POINTS: | ERPROC |
| FORMAL PARAMETERS: | ARGMNT: Argument of QDATA call |
| COMMON BLOCKS: | C10, ERRCOM, OOPS |
| SUBROUTINES CALLED: | ABORT |
| CALLED BY: | QDATA |

Method :

First, the action to be taken (TYPEAC) is set to the normal action (NORMAC). Next, the list of error codes (CHEKS) is checked and if any match to the code causing the call (ERCODE) is found, TYPEAC is set to CHCKAC. The error message is now printed on ERUNIT. The process now depends upon the value in TYPEAC.

If TYPEAC = 'ABORT' the ABORT subroutine is called.

If TYPEAC = 'FLAG' the error flag ERROR is set.

If TYPEAC = 'PASS' no action is taken

If TYPEAC is none of the above values, the value is displayed in a separate error message and ABORT is called.

Subroutine ERPROC is illustrated in figure 15.

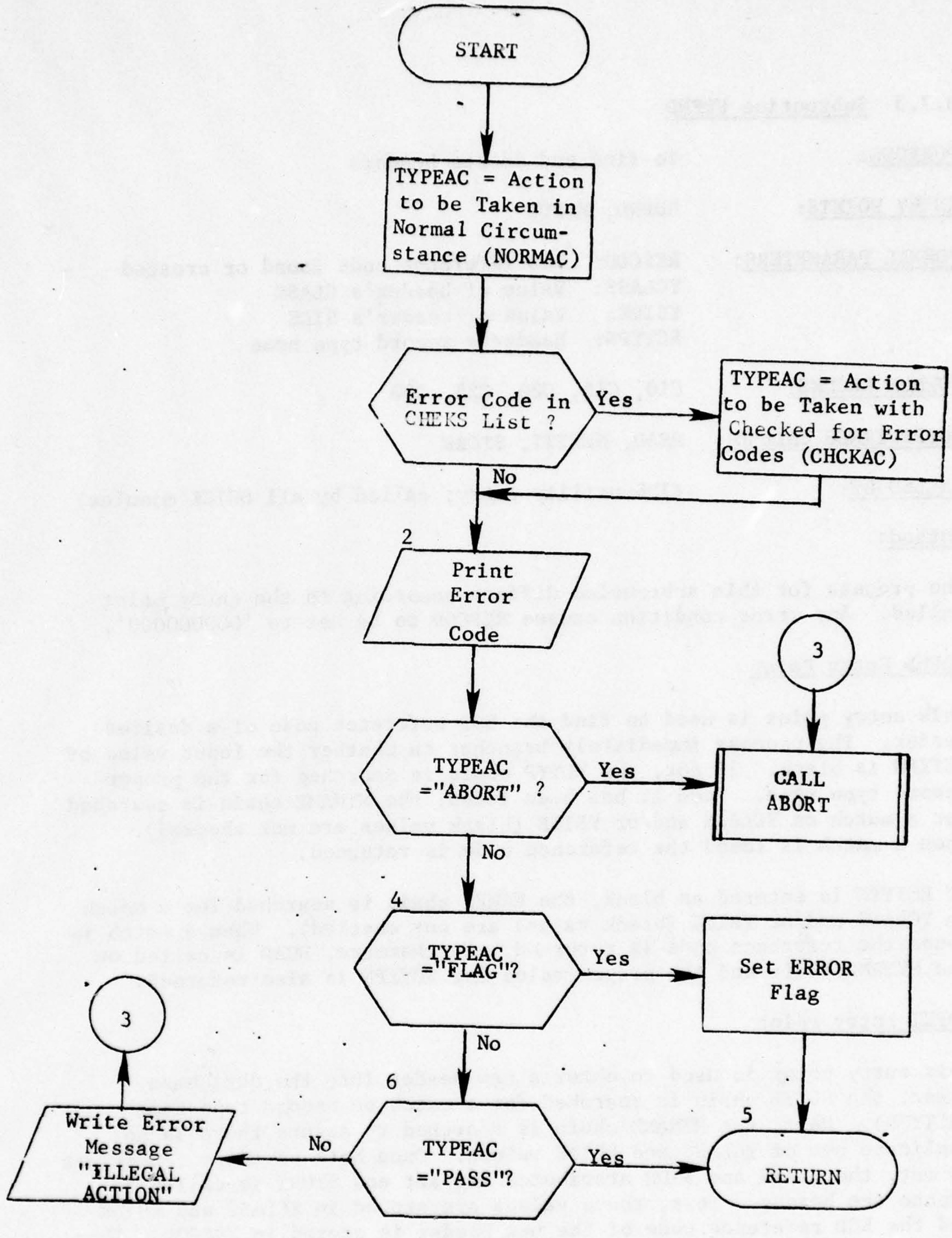



Figure 15. Subroutine ERPROC

3.7.3 Subroutine HDFND

| | |
|---|---|
| PURPOSE: | To find and create headers |
| ENTRY POINTS: | HDFND, HDPUT |
| FORMAL PARAMETERS: | REFCOD: BCD reference code found or created<br>YCLASS: Value of header's CLASS<br>YSIDE: Value of header's SIDE<br>RCTYPN: Header's record type name |
| COMMON BLOCKS: | C10, C15, C20, C25, C30 |
| SUBROUTINES CALLED: | HEAD, NEXTTT, STORE |
| CALLED BY: | (IDS utility entry; called by all QUICK modules) |

Method:

The process for this subroutine differs according to the entry point called. Any error condition causes REFCOD to be set to '000000000'.

HDFND Entry Point

This entry point is used to find the BCD reference code of a desired header. The process immediately branches on whether the input value of RCTYPN is blank. If not, the RCTYP chain is searched for the proper record type name. Once it has been found, the MYNAMZ chain is searched for a match on YCLASS and/or YSIDE (blank values are not checked). When a match is found the reference code is returned.

If RCTYPN is entered as blank, the NAMEZ chain is searched for a match on YCLASS and/or YSIDE (blank values are not checked). When a match is found the reference code is returned. Furthermore, HEAD is called on the MYNAMZ chain and the proper value for RCTYPN is also returned.

HDPUT Entry Point

This entry point is used to enter a new header into the data base. First, the RCTYP chain is searched for a match on record type name (RCTYPN). Next, the MYNAMZ chain is searched to assure there is no duplicate set of YCLASS and YSIDE values. Once both of these conditions is met, the CLASS and SIDE attributes are set and STORE is called to create the header. Next, these values are stored in XCLASS and XSIDE and the BCD reference code of the new header is stored in XREFCD. Then STORE is called to create a new INDHCS record.

Subroutine HDFND is illustrated in figure 16.

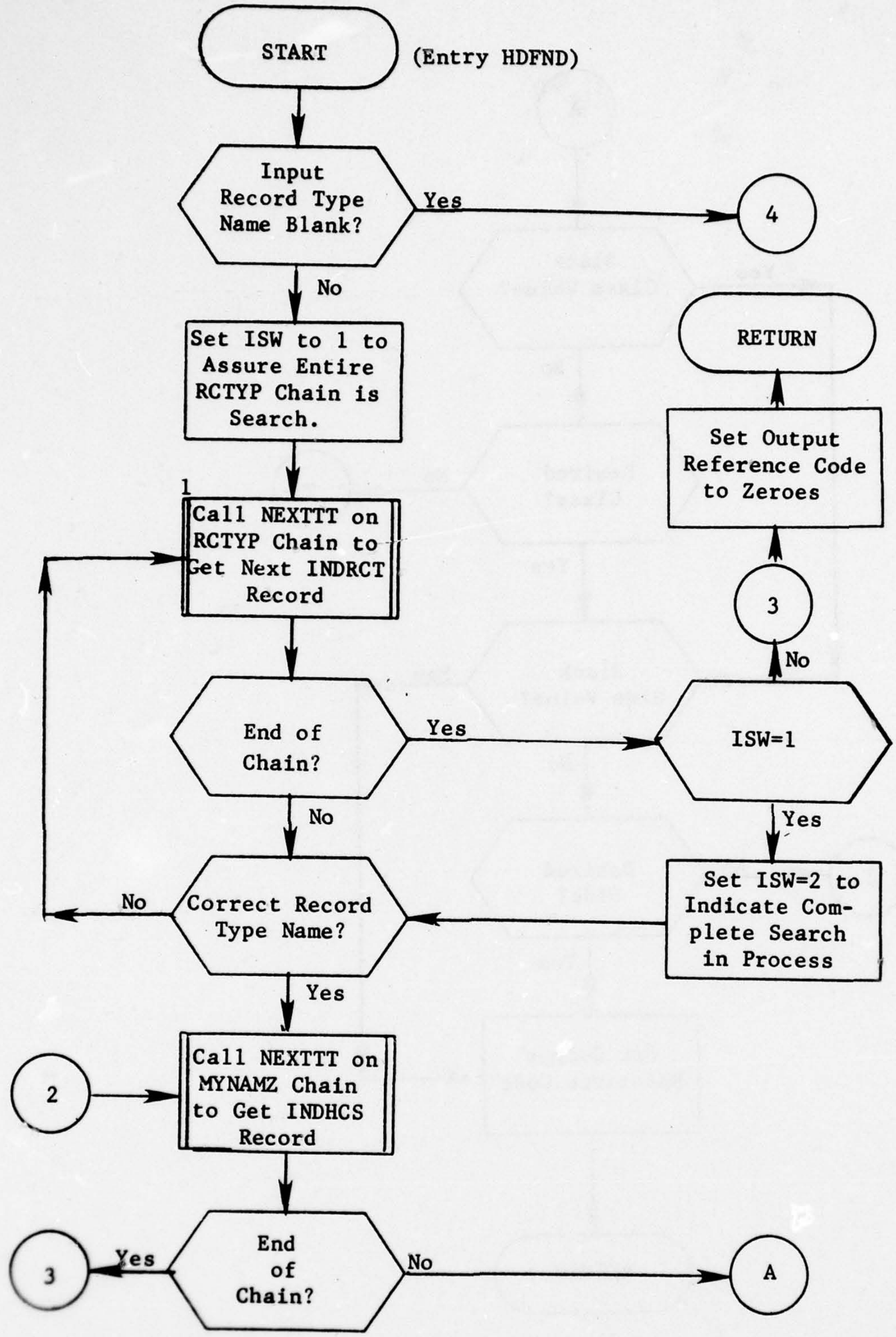



Figure 16. Subroutine HDFND: Entry HDFND (Part 1 of 5)

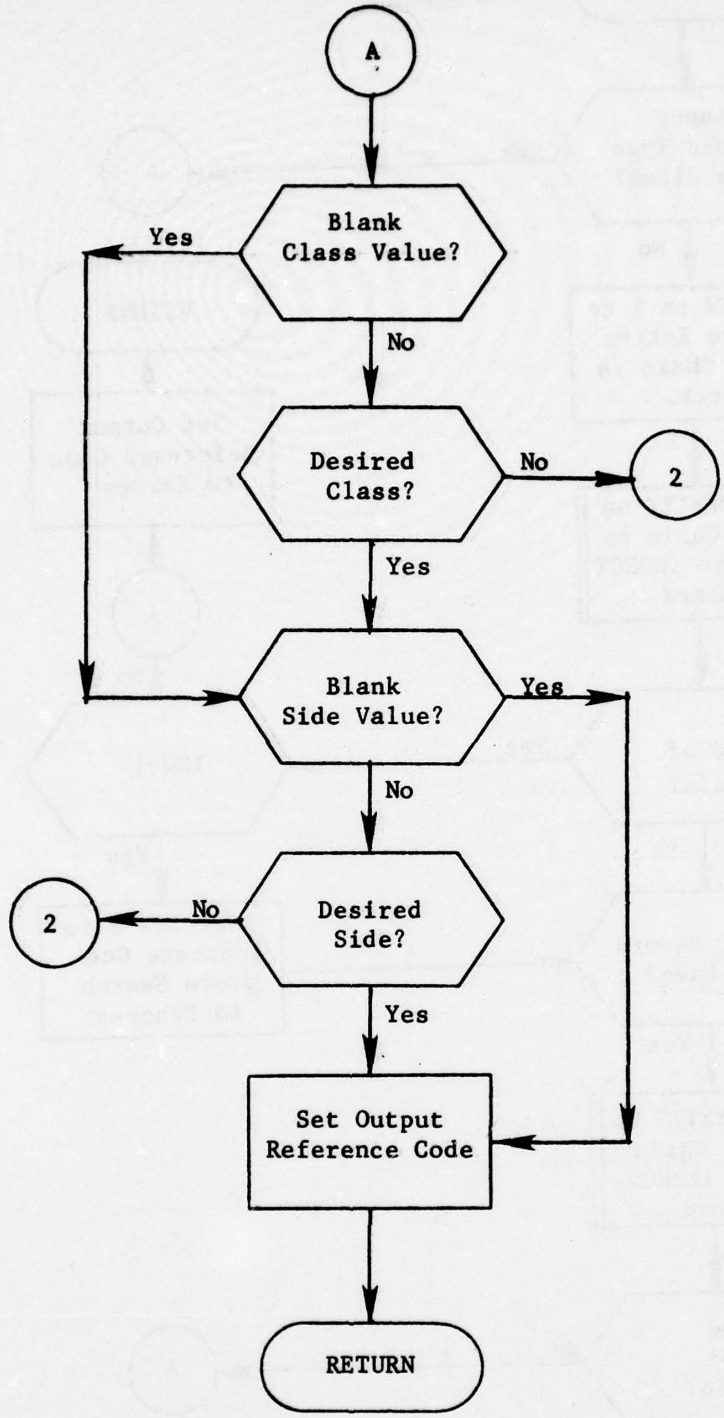



Figure 16. (Part 2 of 5)

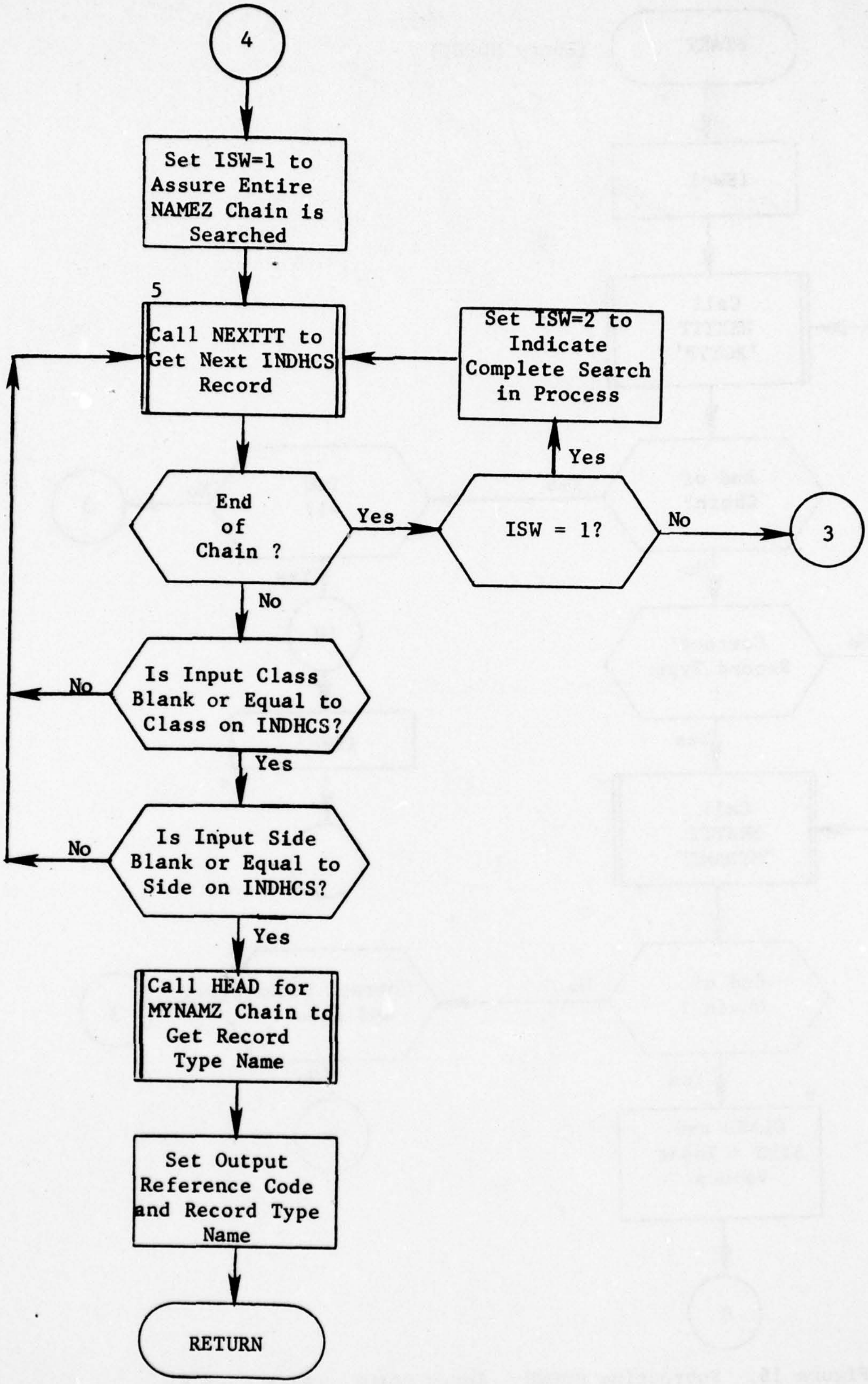



Figure 16. (Part 3 of 5)

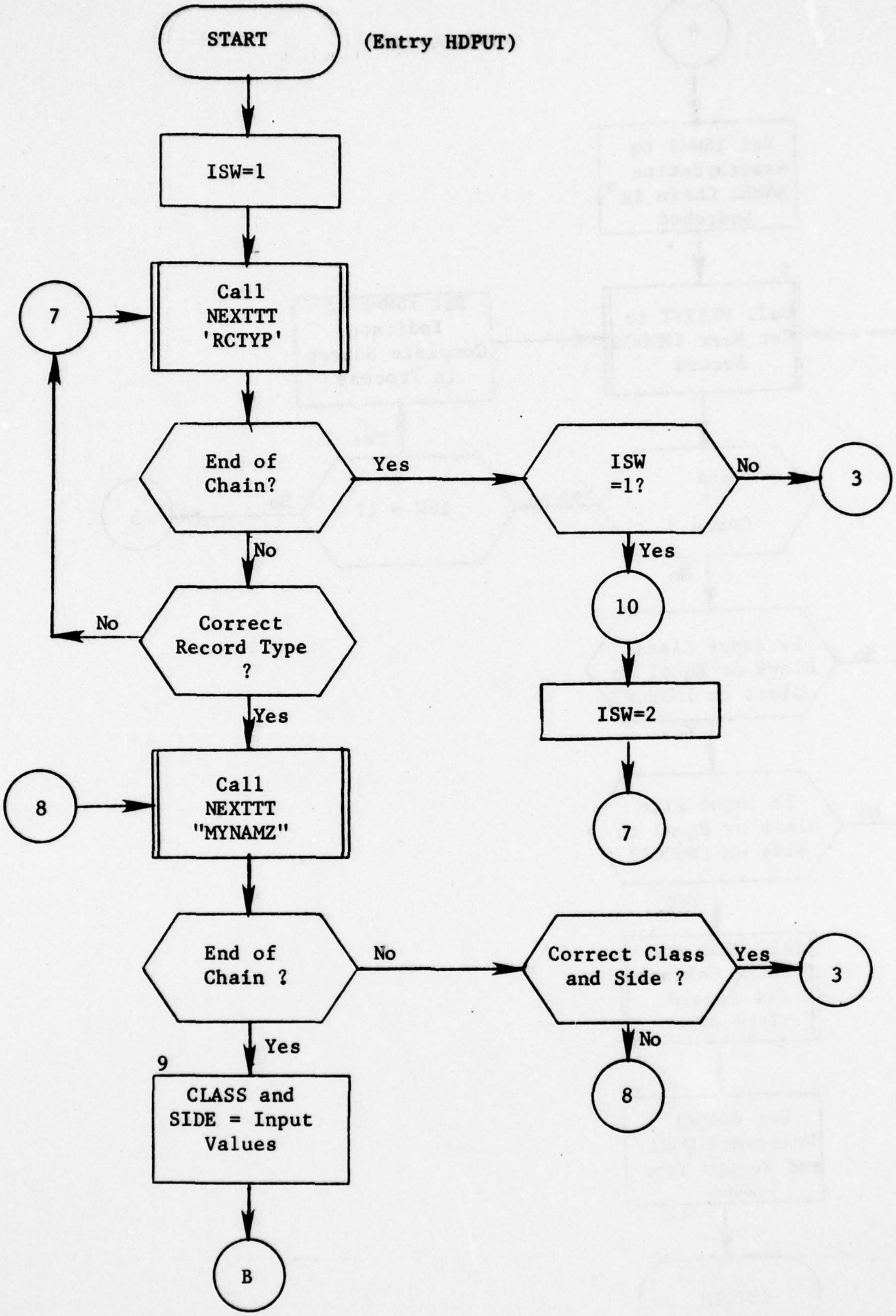



Figure 16. Subroutine HDFND: Entry HDPUT (Part 4 of 5)

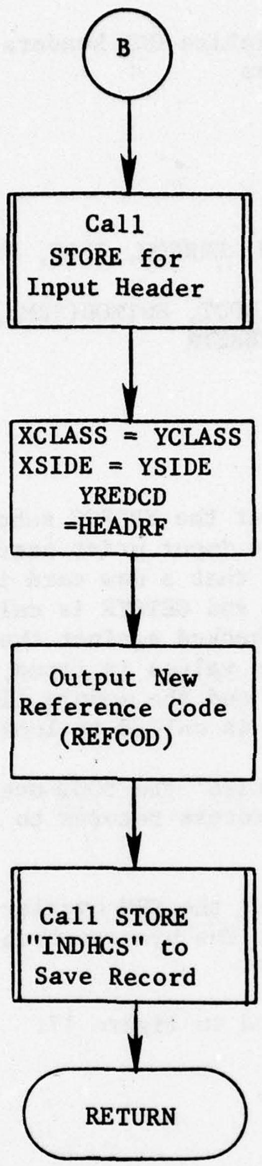



Figure 16. (Part 5 of 5)

3.7.4 Subroutine INICOP

| | |
|---|---|
| PURPOSE: | To initialize COP headers and check for special run modes |
| ENTRY POINTS: | INICOP |
| FORMAL PARAMETERS: | None |
| COMMON BLOCKS: | C15, C30, ERRCOM, IPQT, STRING |
| SUBROUTINES CALLED: | BANNER, BOOT, ENTMOD(SRM), GETSTR, HDFND, OPNIDS, RETRV, WEBSTR |
| CALLED BY: | COP |

Method:

First, the standard settings for the ERPROC subroutine are set. Next the IDS file is opened and the input print header written. Now the input string pointer is set so that a new card image will be read, the ERRFND overlays are brought in and GETSTR is called for the first input string. The first string is checked against the values 'INITIALIZE' and 'RESTORE'. If neither of these values is found, the utility table, index, and dictionary headers, and the module link table are retrieved. Finally, the WEBSTR subroutine is called to look up the input string.

If the first string is 'INITIALIZE' the BOOT overlay is executed. Following its execution, the process returns to get the next string and retrieve the headers.

If the first string is 'RESTORE' the SRM overlay is executed after setting ERROR on as a flag to SRM. The process then returns to get the next string and retrieve the headers.

Subroutine INICOP is illustrated in figure 17.

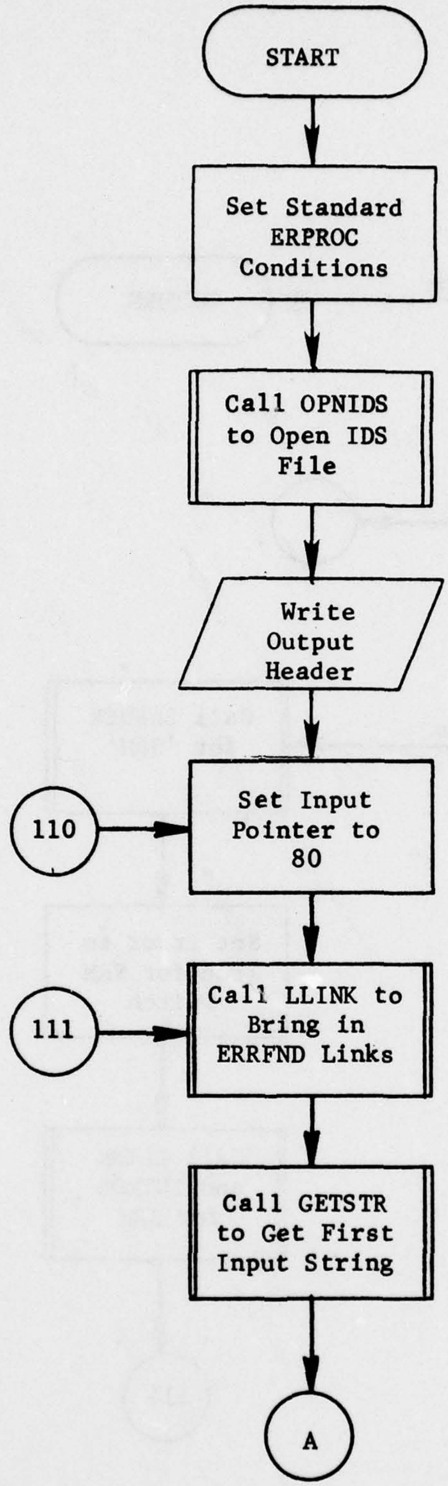



Figure 17. Subroutine INICOP (Part 1 of 3)

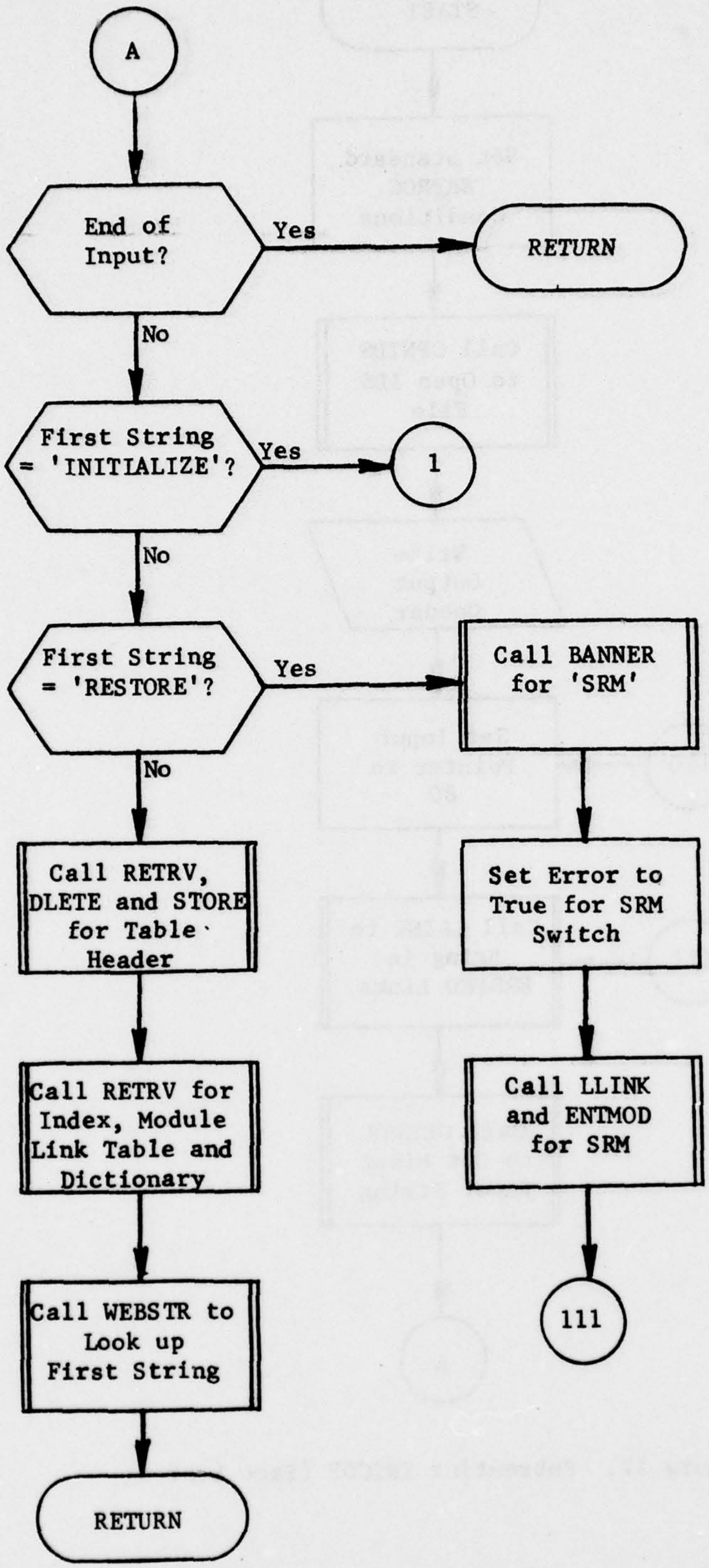



Figure 17. (Part 2 of 3)

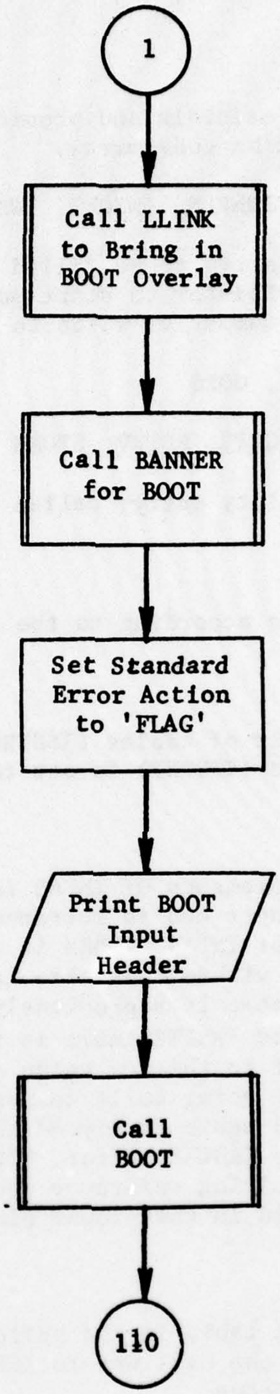



Figure 17. (Part 3 of 3)

3.7.5 Subroutine INSPUT

| | |
|---|---|
| PURPOSE: | Create, maintain and provide values from the input instruction code array. |
| ENTRY POINTS: | INSDEL, INSFLS, INSGET, INSNIT, INSPUT |
| FORMAL PARAMETERS: | INTAB: Array to be filled or used to fill table<br>INDEX: Pointer to start point in table for process<br>LENTH: Number of words to be stored or retrieved |
| COMMON BLOCKS: | C40, INS, OOPS |
| SUBROUTINES CALLED: | DLETE, MODFY, RETRV, STORE |
| CALLED BY: | *(IDS utility entry; called by all QUICK modules)* |

Method:

This subroutine process differs according to the entry point called.

INSNIT Entry Point

This entry point sets the number of tables (INSTBS) to zero. Also, the array for table reference codes (INSREF) is set to blanks.

INSPUT Entry Point

This entry point enters LENTH elements of INTAB in the Instruction Code array. INDEX is used as a pointer and is incremented prior to each entry. Beginning with the first word of INTAB, INDEX is incremented and the table number and pointer to the word within the table is calculated (each table contains 100 words). If the table is a previously created table, the subroutine assures that the desired TABLEZ table is in C40. Once this is done, the indicated word is set to the new value and MODFY is called. If the desired table is the one being built currently the new value is simply entered. If the desired table is beyond the current table, the current table is moved into the TABLEZ buffer, STORE is called to create a new table and the resulting reference code is saved in INSREF. Finally, the new value is stored in the proper place.

INSFLS Entry Point

This entry point saves the last table in the buffer. The method is to simply set a switch indicating the call was to INSFLS and branch to the appropriate part of the INSPUT code.

INSGET Entry Point

This entry point retrieves from the Instruction Code Array and stores the values in INTAB. The process is a loop of length LENTH. For each pass, NDEX, originally set to INDEX-1, is incremented. From NDEX, the table and index within table is calculated. If the desired table is not in the buffer it is retrieved and moved into the buffer. The index is then used to set the value into INTAB. If in the process NDEX goes beyond the available tables, the value 1 is set into INTAB.

INSDEL Entry Point

This entry point retrieves all TABLEZ's in succession and deletes them.

Subroutine INSPUT is illustrated in figure 18.

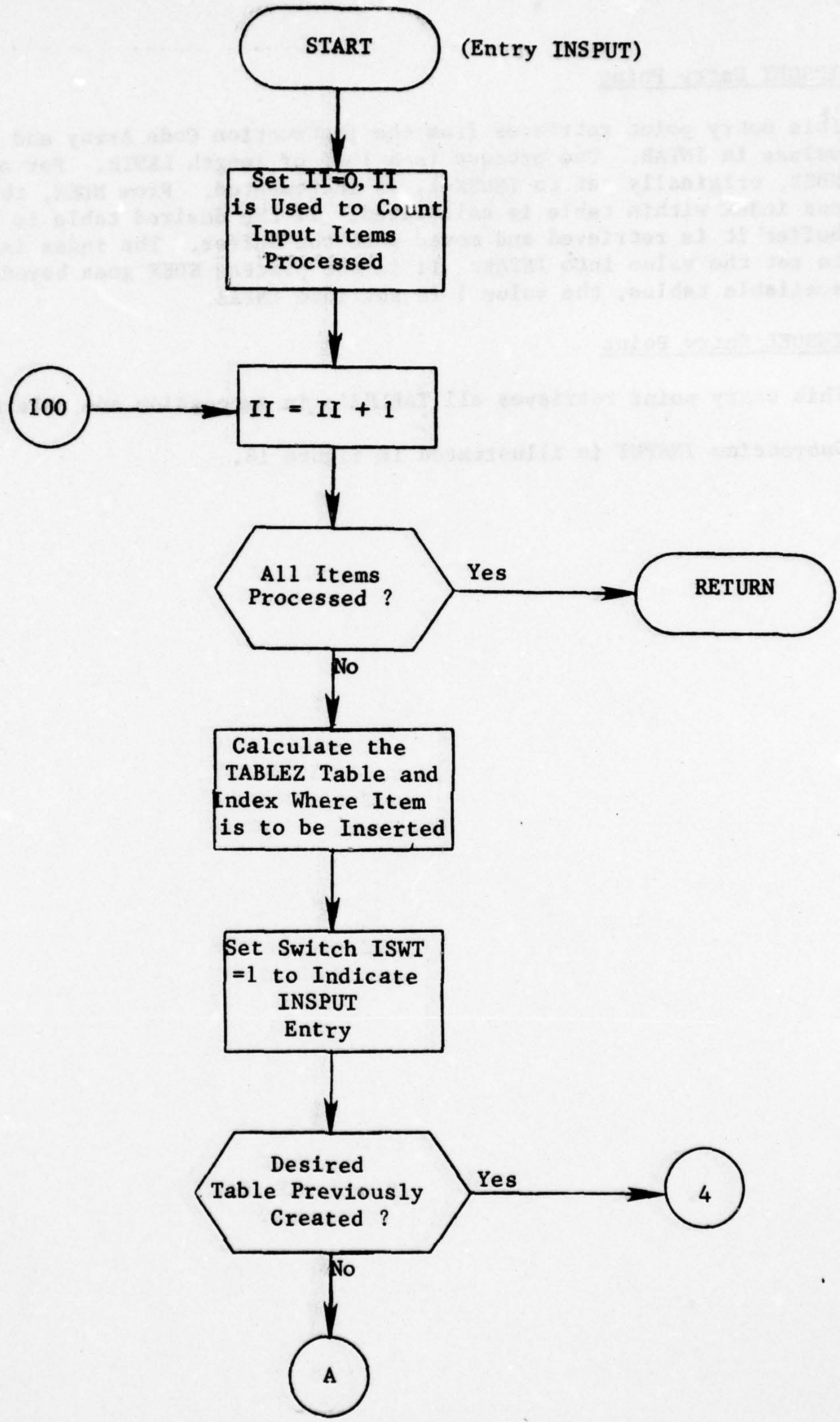



Figure 18. Subroutine INSPUT: Entry INSPUT (Part 1 of 7)

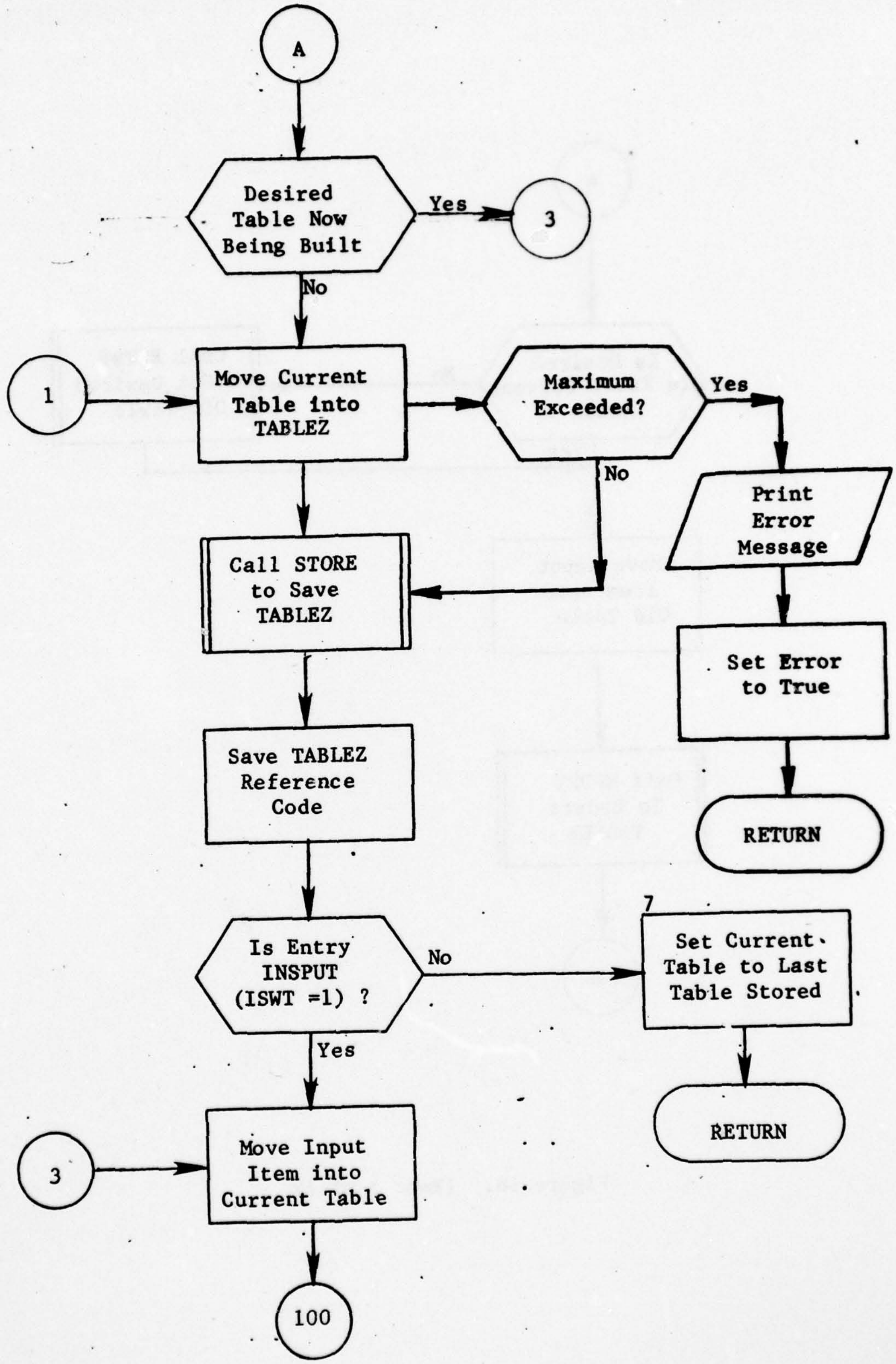



Figure 18. (Part 2 of 7)

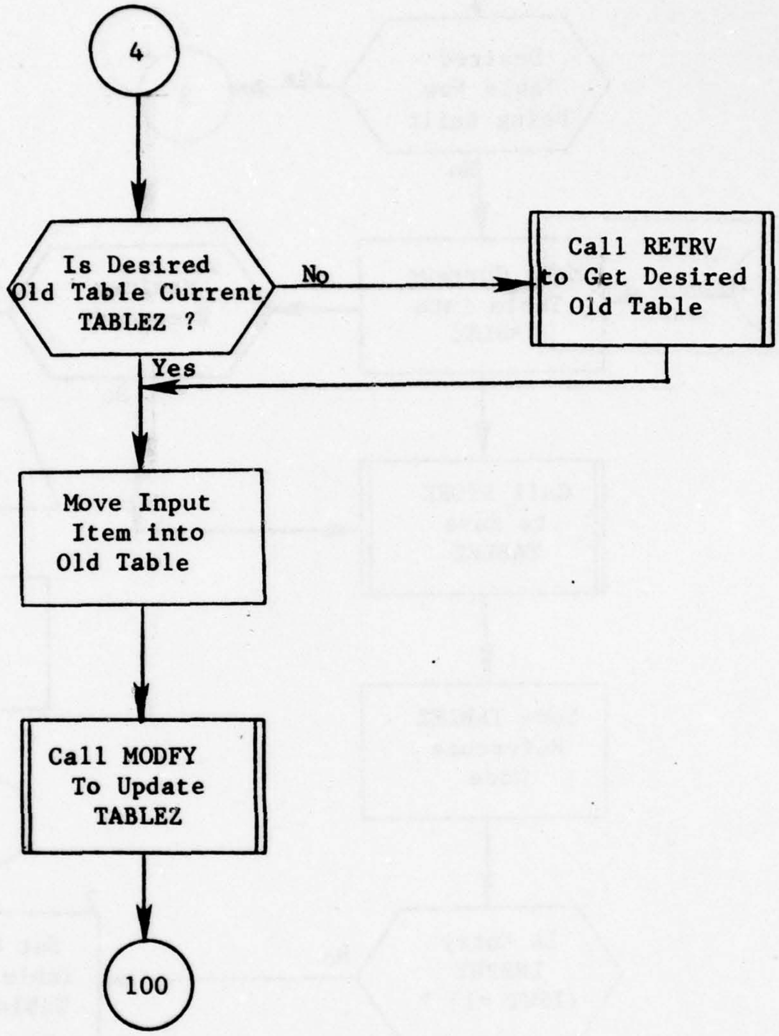



Figure 18. (Part 3 of 7)

AD-A054 377 COMMAND AND CONTROL TECHNICAL CENTER WASHINGTON D C F/G 15/7
THE CCTC QUICK-REACTING GENERAL WAR GAMING SYSTEM. (QUICK). PRO--ETC(U)
JUN 77 D J SANDERS, P F MAYKRANTZ, J M HERRIN
UNCLASSIFIED CCTC-CSM-MM-9-77-V1-PT-1 SBIE-AD-E100 051 NL

2 OF 5
AD
A054377

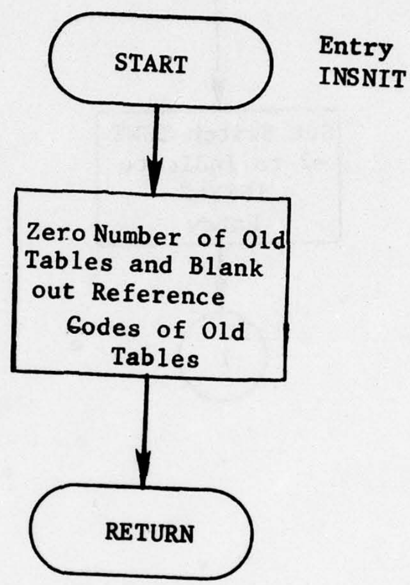



Figure 18. Entry INSNIT (Part 4 of 7)

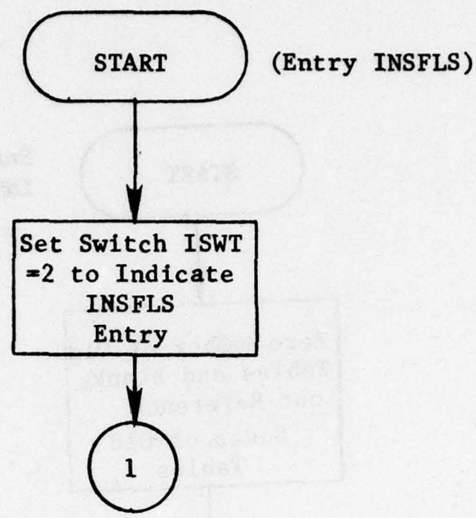



Figure 18. Subroutine INSPUT: Entry INSFLS (Part 5 of 7)

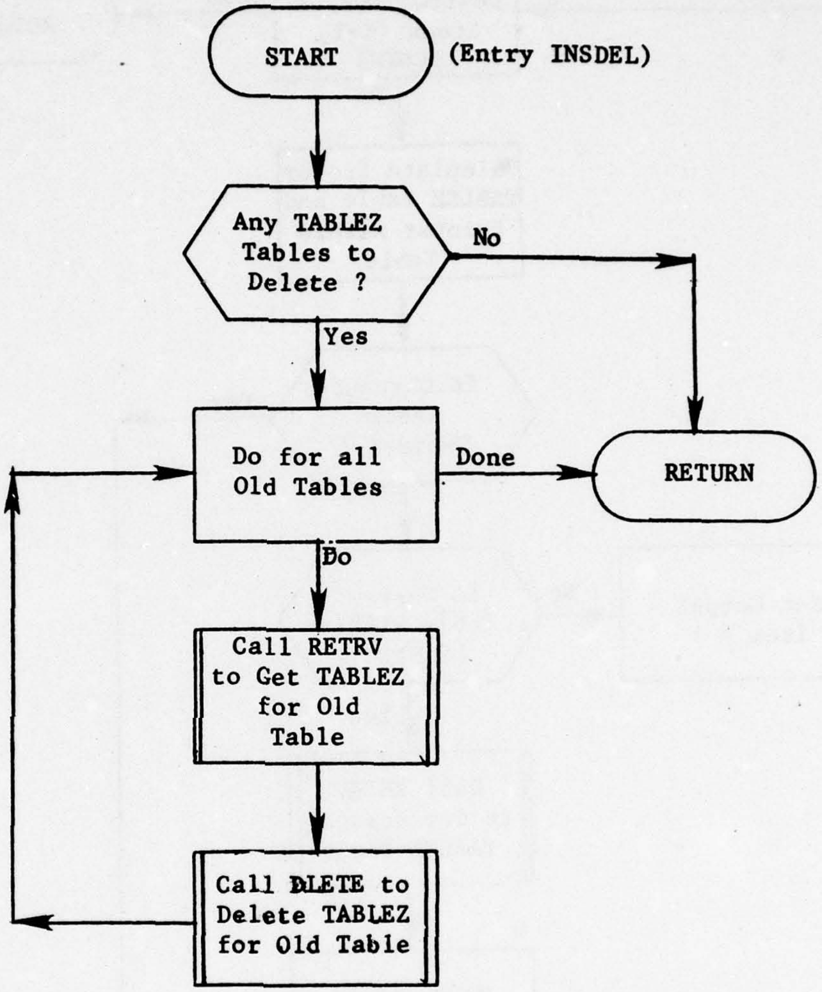



Figure 18. Subroutine INSPUT: Entry INSDEL (Part 6 of 7)

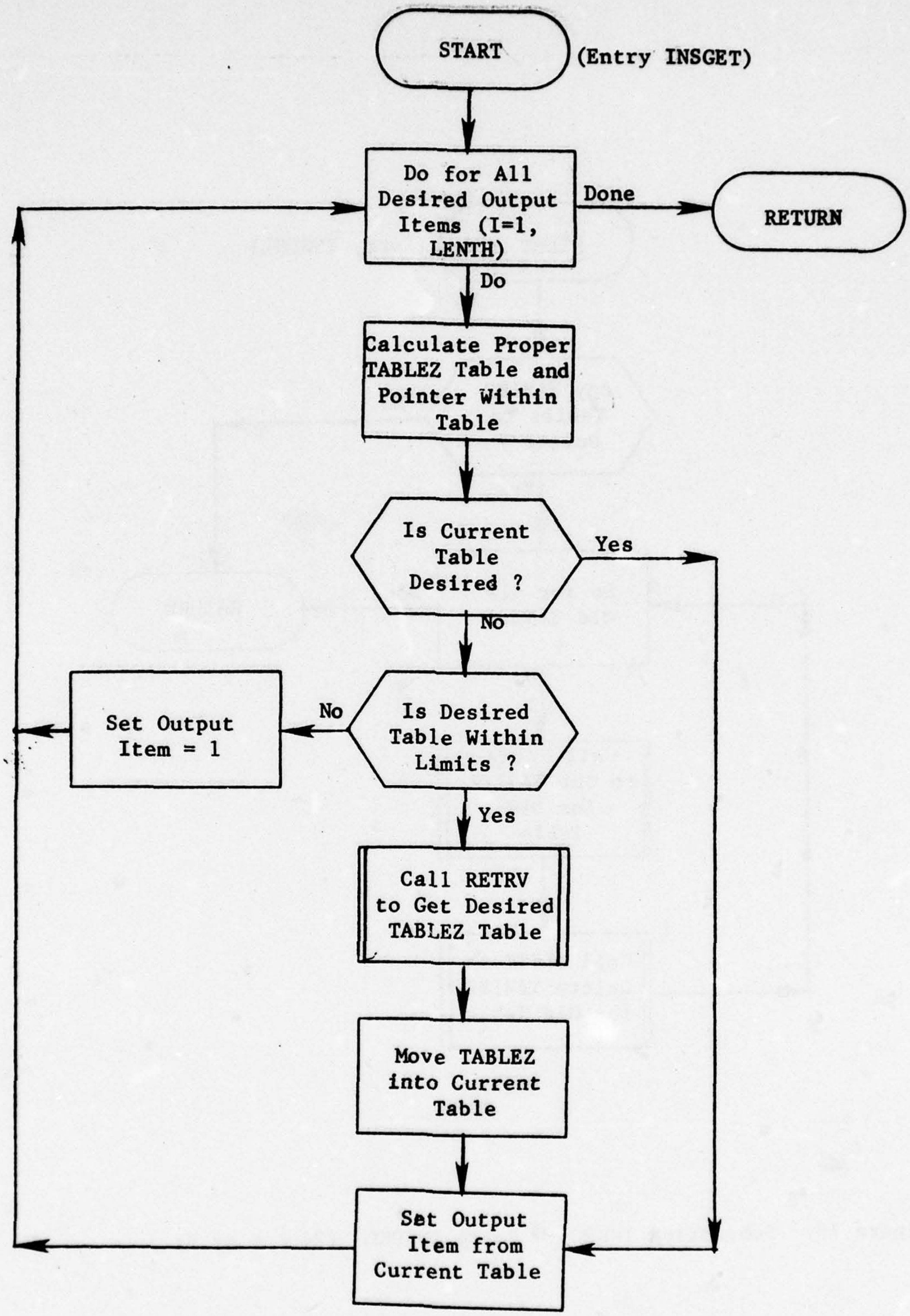



Figure 18. Subroutine INSPUT: Entry INSGET (Part 7 of 7)

### 3.7.6 Subroutine MODGET

| | |
|---|---|
| PURPOSE: | Execute modules |
| ENTRY POINTS: | MODGET |
| FORMAL PARAMETERS: | None |
| COMMON BLOCKS: | C35 |
| SUBROUTINES CALLED: | BANNER, ENTMOD, INSGET |
| CALLED BY: | COP |

Method:

First INSGET is called to get the verb's number. This number is used as an index to the module link table (common block C35) to obtain an overlay link name. This name (NEWMOD) is compared to the old name (OLDMOD). If they are different, BANNER is called to display NEWMOD. OLDMOD is set to NEWMOD. System routine LLINK is called to read in overlay NEWMOD. Finally, standard module entry point ENTMOD is called.

Subroutine MODGET is illustrated in figure 19.

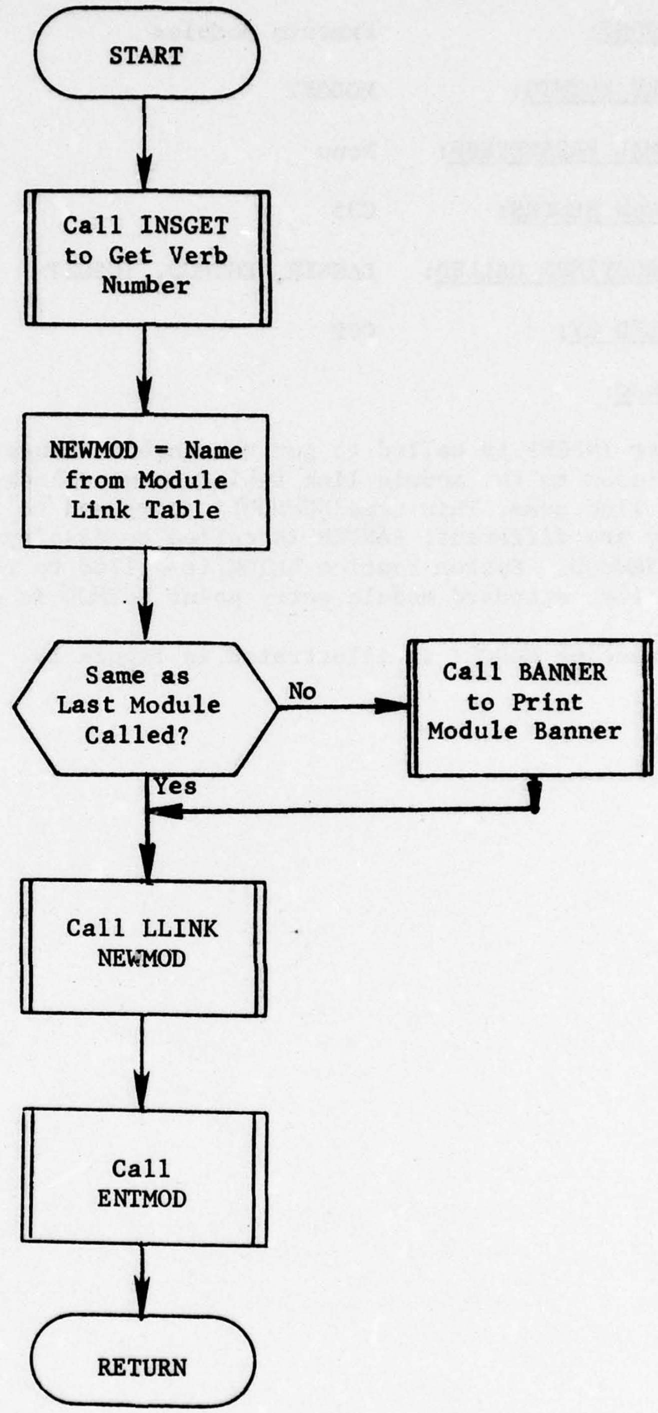



Figure 19. Subroutine MODGET

### 3.7.7 Subroutine QDATA

| | |
|---|---|
| PURPOSE: | Performs IDS functions |
| ENTRY POINTS: | CLZIDS, DIRECT, DLETE, HEAD, MODFY, NEXTTT, OPNIDS, RETRV, STORE |
| FORMAL PARAMETERS: | ARGUMENT: Record type or chain name |
| COMMON BLOCKS: | C10, C15, C20, C25, C30, C35, C40, C50 |
| SUBROUTINES CALLED: | ERPROC |
| CALLED BY: | (IDS utility entry, called by all QUICK modules) |

Method:

The entry points for this routine fall into three groups: those with no argument: CLZIDS, OPNIDS, DIRECT; those whose argument is a record type name: DLETE, MODFY, RETRV, STORE; and those whose argument is a chain name: HEAD, NEXTTT.

CLZIDS, OPNIDS and DIRECT

These entries each carry out a single function. CLZIDS closes the IDS file. OPNIDS opens the IDS file for update. DIRECT retrieves the record whose binary reference code is stored in C10.

DLETE, MODFY, RETRV and STORE

For each of these entry points the process is quite similar. The input record type name is looked up in a table of legal record type names (REC-TYPE-TABLE). If it is not in the table the error code is set to 'OOORRR'. After the look-up, the subroutine branches to the appropriate code to perform the requested function. In some cases (see notes to figure 20) the requested function is not allowed in which event the rror code is set to 'OOOILC'.

HEAD and NEXTTT

For each of these entry points the process is similar. The input chain name is looked up in a table of legal chain names (CHAIN-NAME-TABLE). If it is not in the table, the error code is set to 'OOOCCC'. After the look-up, the subroutine branches to the appropriate code to perform the requested function.

After any of the above entry points process have taken place, the error code is checked. If an error has occurred ERPROC is called.

Subroutine QDATA is illustrated in figure 20.

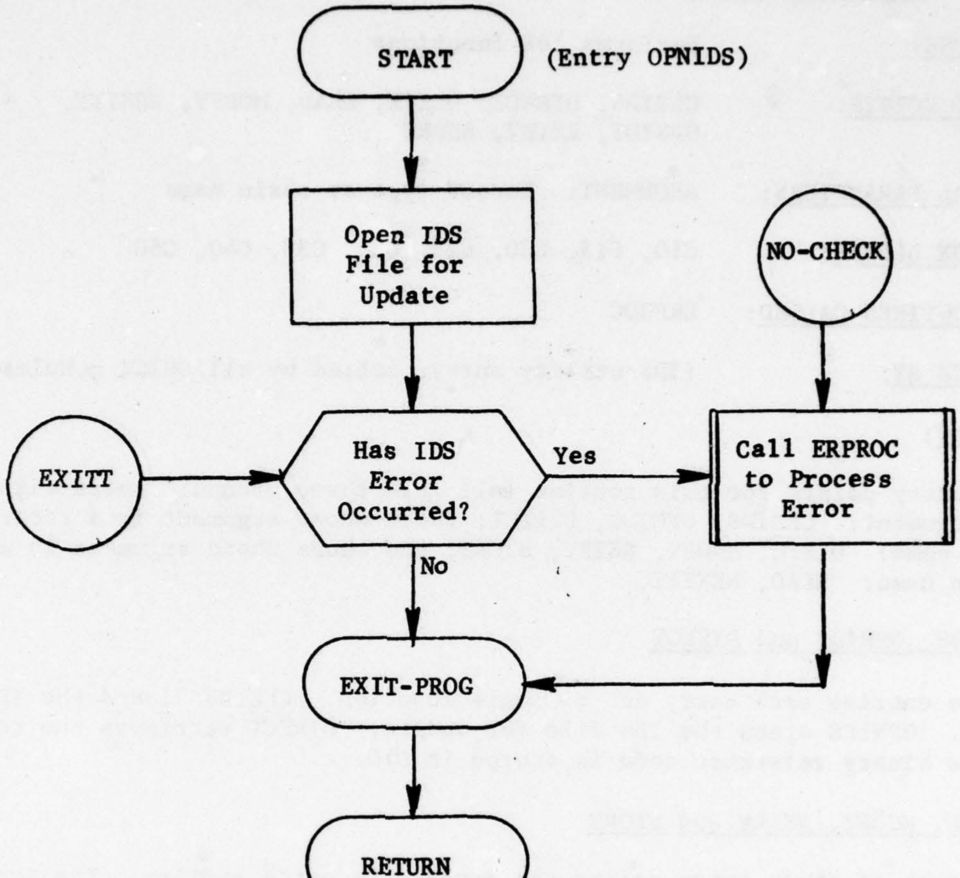



Figure 20. Subroutine QDATA: Entry OPNIDS (Part 1 of 9)

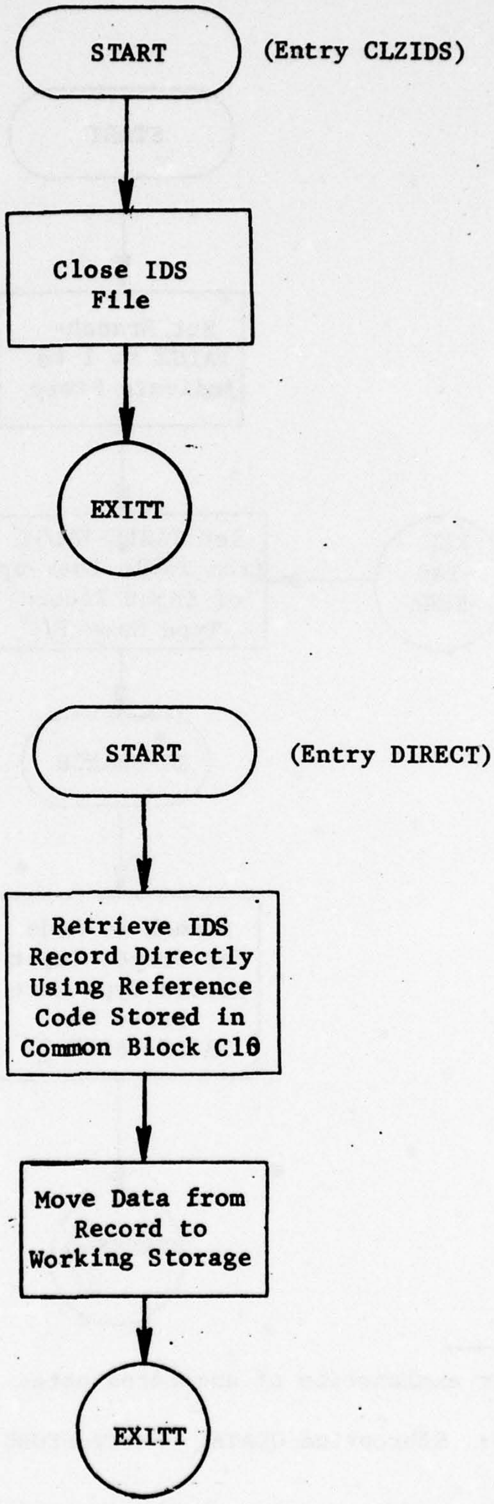
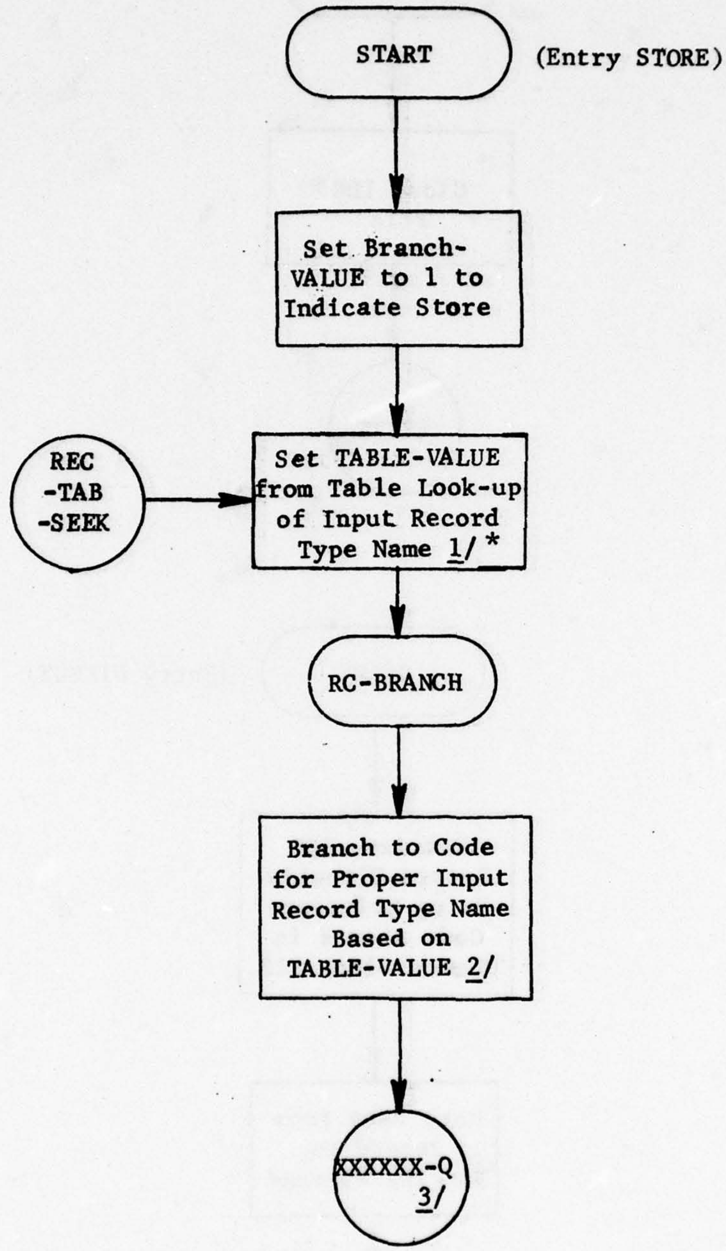



Figure 20. Subroutine ODATA: Entry CLZIDS and DIRECT (Part 2 of 9)




* See part 9 for explanation of annotated notes

Figure 20. Subroutine QDATA: Entry STORE (Part 3 of 9)

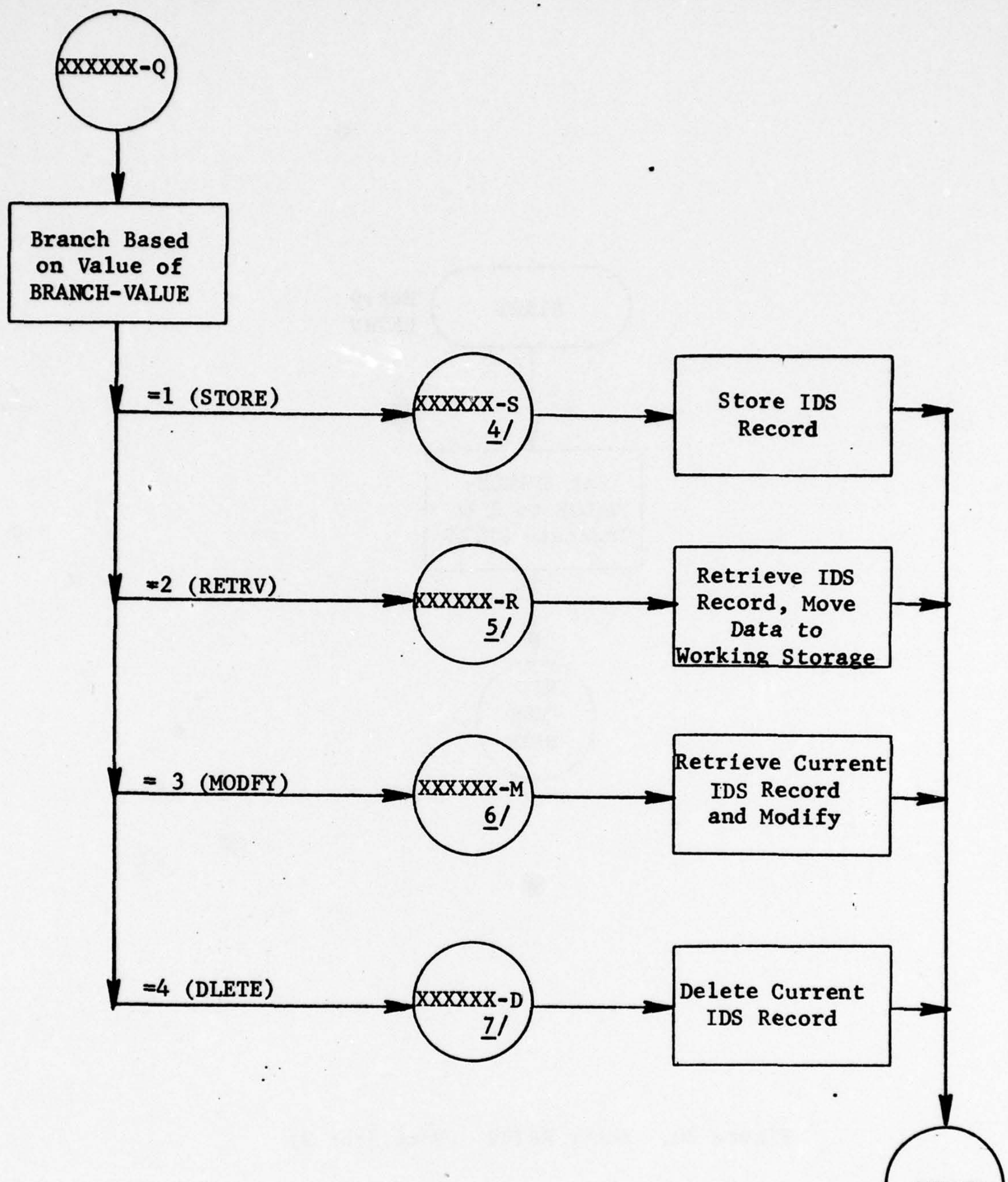



Figure 20. (Part 4 of 9)

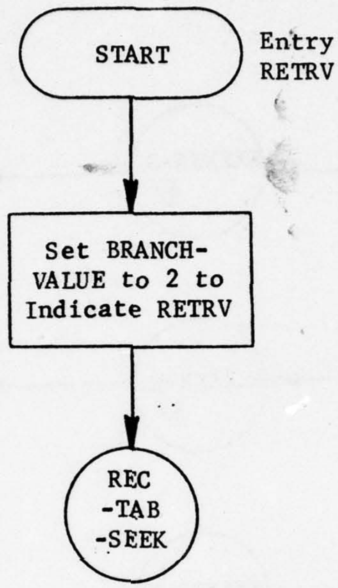



Figure 20. Entry RETRV (Part 5 of 9)

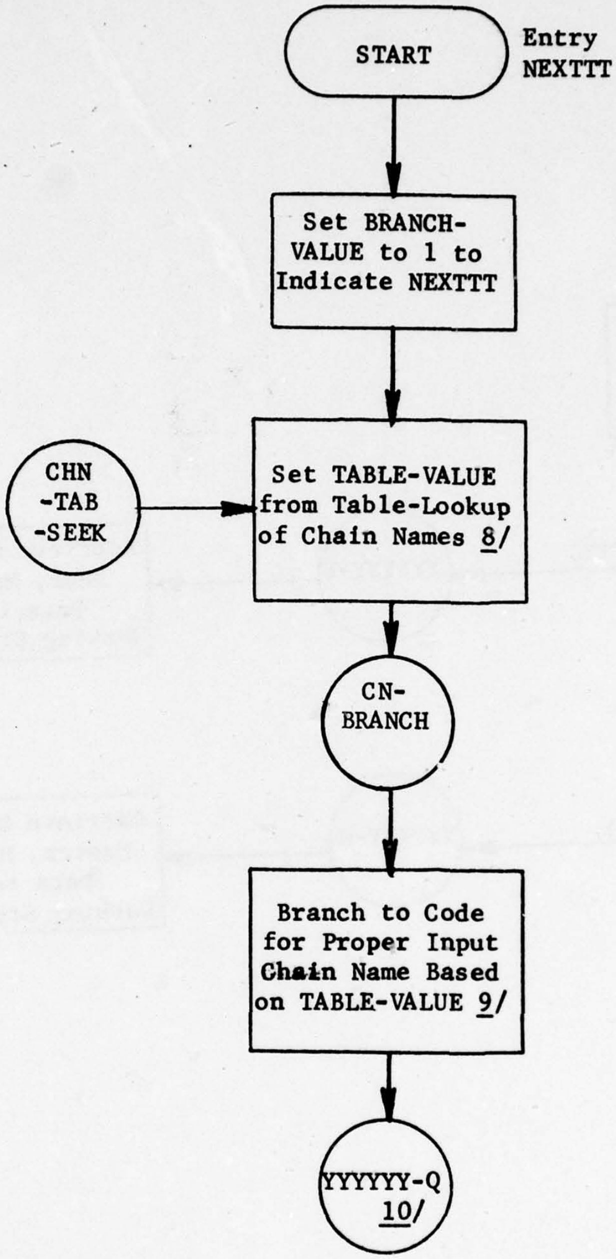



Figure 20. Entry NEXTTT (Part 6 of 9).

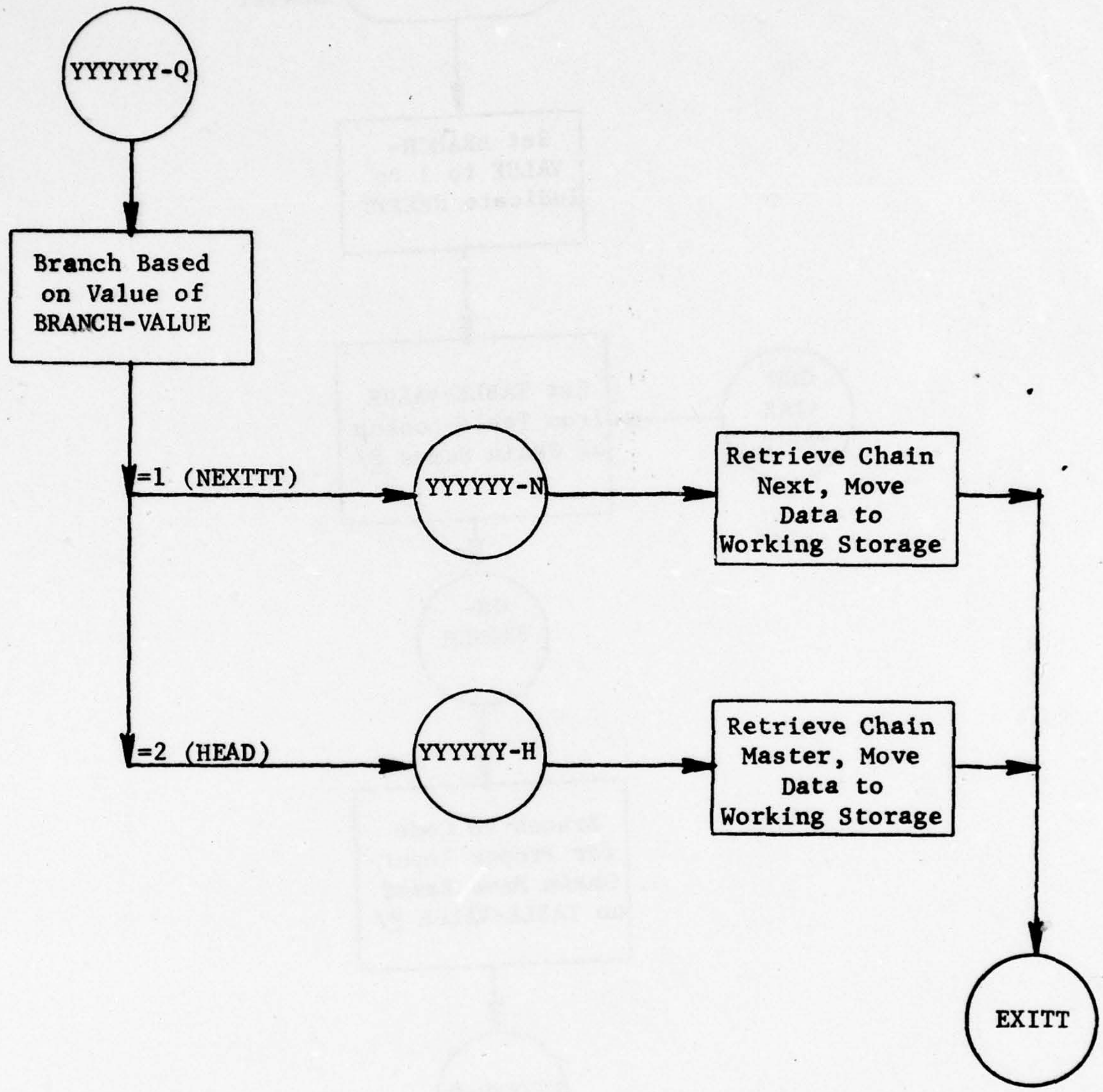



Figure 20. Entry NEXTTT (Part 7 of 9)

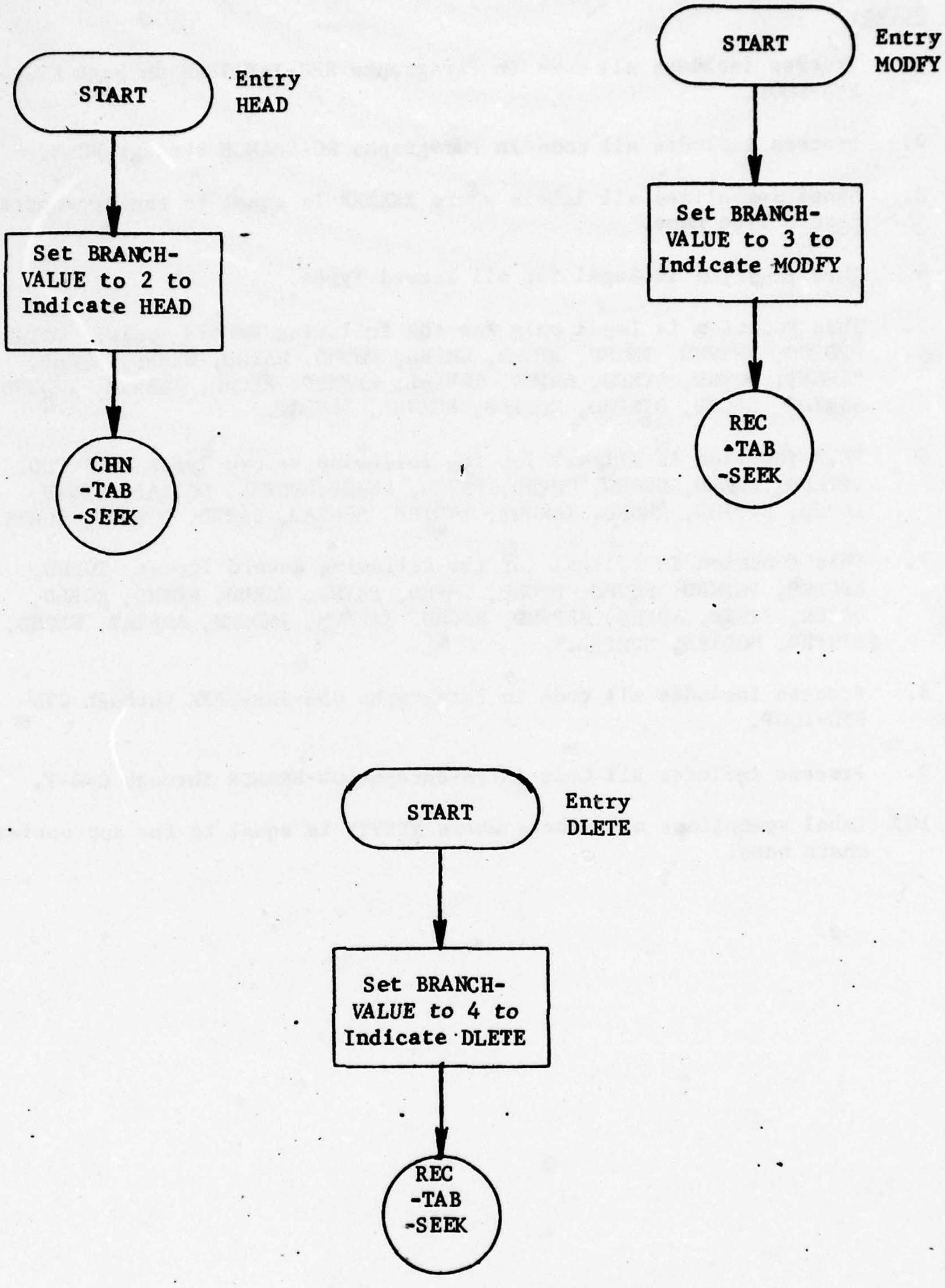



Figure 20. Entries HEAD, MODFY, and DLETE
(Part 8 of 9)

Notes:

1. Process includes all code in Paragraphs REC-TAB-SEEK through RTS-END-LOOP.

2. Process includes all code in Paragraphs RC-BRANCH through RC-X.

3. Label Symbolizes all labels where XXXXXX is equal to the appropriate Record Type name.

4. This function is legal for all Record Types.

5. This function is Legal only for the following Record Types: TGTHD, WPGPHD, VNTHHD, PNCHD, DPCHD, CMPHD, PAYHD, WARHD, WEPHD, RCBHD, TARGET, DCTHD, SYNHD, ADVHD, TABLEZ, RFPTHD, REGHD, TARNUM, INDTHD, ASNTAB, SECHD, DISPHD, MODTAB, NUMTBL, TABLST.

6. This function is illegal for the following Record Types: WPGPHD, VNTKHD, PNCHD, DPCHD, CMPHD, PAYHD, REBHD, DCTHD, DCTTAB, SYNHD, ADVHD, RFPTHD, REGHD, TARNUM, INDTHD, ASNTAB, SECHD, DISPHD, SORTAB.

7. This function is illegal for the following Record Types: TGTHD, WPGPHD, VNTKHD, PNCHD, DPCHD, CMPHD, PAYHD, WARHD, WEPHD, RCBHD, DCTHD, SYNHD, ADVHD, RFPTHD, REGHD, TARNUM, INDTHD, ASNTAB, SECHD, DISPHD, MODTAB, NUMTBL.

8. Process includes all code in Paragraphs CHN-TAB-SEEK through CTS-END-LOOP.

9. Process includes all Code in Paragraphs CN-BRANCH through CNB-F.

10. Label symbolizes all labels where YYYYYY is equal to the appropriate chain name.

Figure 20. (Part 9 of 9)

## 3.8 Subroutine BOOT*

**PURPOSE:** Create and update organizational data

**ENTRY POINTS:** BOOT

**FORMAL PARAMETERS:** None

**COMMON BLOCKS:** C10, C15, C20, C30, C35

**SUBROUTINES CALLED:** DCTFND, HDFND, HDPUT, HEAD, MNMFND, MODFY, NEXTTT, NUMFND, RETRV, RNMFND, SEEKER, STORE, STRMAK

**CALLED BY:** INICOP

**Method:**

BOOT reads card images which instruct it as to what actions to take. The card images are in sets which are begun by an introductory adverb and ended with an END card. The first part of the process is to read an adverb (NEWINDEX, RECORDTYP, INDEX, DICTIONARY, SYNTAX, MODULE, HEADER, SECTORS). Each adverb causes the branch IBR to be set to a different value. If an adverb is read which is not recognized processing ceases. The method used for each adverb is different. However, each card image that is read is printed after any action is taken with an appropriate flag (IND).

**NEWINDEX**

No command cards follow this abverb. The action taken is to set the CLASS attribute and call STORE to create the utility table and index headers.

**RECORDTYP**

Each card image creates a new INDRCT record. The process is to call NUMFND for the record type number then search the RCTYP chain for a match. If a match is found, IND is set for ignored input. If no match is found, IND is set to show added record and STORE is called to create an INDRCT record.

**INDEX**

Each card image creates a new record of the type specified in the first field of the card image.

---

*Main routine of overlay BOOTT.

INDMST or INDDET: RNMFND is called for the record type numbers and to check the validity of both record type names. SEEKER is then called on the appropriate chain (IRMAST or IRDET, respectively) to look for a duplicate. If a duplicate is found the ignored flag (IND) is set. Otherwise, the appropriate attributes are set and STORE is called.

INDATR: First STRMAK is called to obtain the attribute name. Then this name is checked for validity by DCTFND. Next the ATRIB chain is queried by SEEKER. If a match is found IND is set to indicate a change, if not it is set to indicate an add. Now MNMFND is used to get the value of the attribute type. This type is used to determine how to read the default and range fields. Finally, depending upon IND, either STORE or MODFY is called.

ALPHVL

First STRMAK is called to obtain the attribute name, and the name is validated by DCTFND. Next SEEKER is used to find this attribute on the ATRIB chain. Then SEEKER is used to check the VALIST chain for a duplicate value. Finally, if all decks are correct, STORE is called to add a ALPHVL record.

LINKER

First STRMAK is called to obtain the attribute name and it is validated by DCTFND. Next RNMFND is used to validate the record type name. Next the IALINK chain is searched by SEEKER for a duplicate, which, if found, causes the flag (IND) to be set for a change. Then MNMFND is called to obtain the value of the control mnemonic. Finally, either MODFY or STORE is called depending upon the flag (IND).

DICTIONARY

Each card image creates a new entry in the dictionary. First STRMAK is called to obtain the input value for the word to be entered in the dictionary. Next DCTFND is called to look for the new word in the dictionary. If the word is found the indicator flag (IND) is set for a change. If neither the word nor its tab character (tab character is formed from the first two characters of the word) is found, STORE is called to create an appropriate tab character record (DCTTAB). Now MNMFND is called to set the word type. If the type is attribute (Type=6) NUMFND is called for the value, and the address and MNMFND is called for the type and identifier flag. These quantities are packed into WORDVL. If not an attribute, NUMFND is called for WORDVL. Finally, either MODFY or STORE is called.

SYNTAX

Each card image creates a new record of the type specified in the first field of the input card.

SYNVRB: STRMAK is called to obtain the verb and DCTFND is used to validate it. SEEKER is then used to look for a match on the VERB chain and IND set to change if one is found. Next MNMFND is called to obtain the value of the clause switch (ICSW). Finally, either MODFY or STORE is called.

PRMADV: STRMAK is called to obtain the adverb name and DCTFND is called to validate it. SEEKER is then used to look for a match on the ADVADV chain and the flag set to change if one is found. Next MNMFND is called for the clause type (IXTYP) and the phrase type (ILTYP). Finally, either MODFY or STORE is called.

ADVELM: STRMAK is called to obtain the adverb name and DCTFND is called to validate it. Next SEEKER is called to find the adverb on the ADVADV chain. MNMFND is now used to find the element's type. If the type is special word (Type=5), or operator (Type=1) DCTFND is used to find the value. Operators are also checked against a list of special characters. Finally, STORE is called for the ADVELM record.

SYNCLZ: First STRMAK is called to obtain the verb and DCTFND is called to validate. Similarly, STRMAK and DCTFND are called for the adverb. Next SEEKER is used to assure that no duplicate SYNCLZ record exists and then is used to find the adverb on the ADVADV chain. Finally, STORE is called.

MODULE

Each card image calls for an entry to the module link table. This table is retrieved at the first call to this section but is not modified until BOOT is terminating (see figure 21, statement 64). For each card image STRMAK is called to obtain the verb name and DCTFND is used to get the verb number. The link name is then stored in LINKS indexed by the verbs number. A valid card for this section is always marked as a change.

HEADER

Each card image calls for a new header to be created. First, RNMFND is called to validate the record type name. Next, HDFND is called to make sure no duplicate exists. If the record type name is TGTHD, NUMFND is called to set ICLASS. Finally, HDPUT is called to create the header.

## SECTOR

Each card image creates a new SECTR record. First, NUMFND is called to obtain upper and lower longitude. These numbers are checked to assure upper is greater than lower. The SECTOR chain is now searched until either an old sector is found which would contain the new sector, or the end of the chain is reached. In the first case MODFY is called for the old sector. In the second case STORE is called for the new sector.

Subroutine BOOT is illustrated in figure 21.

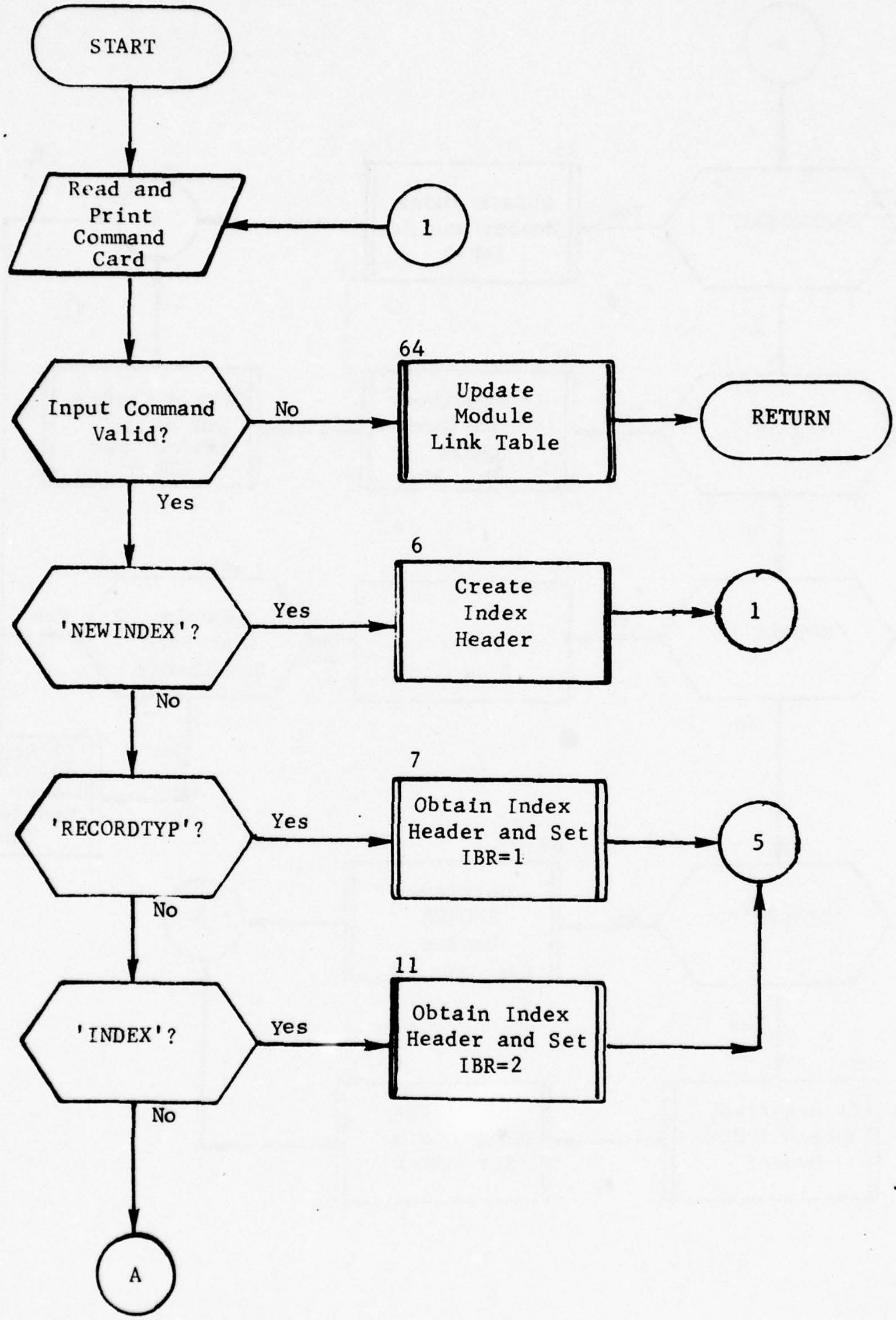



Figure 21. Subroutine BOOT (Part 1 of 26)

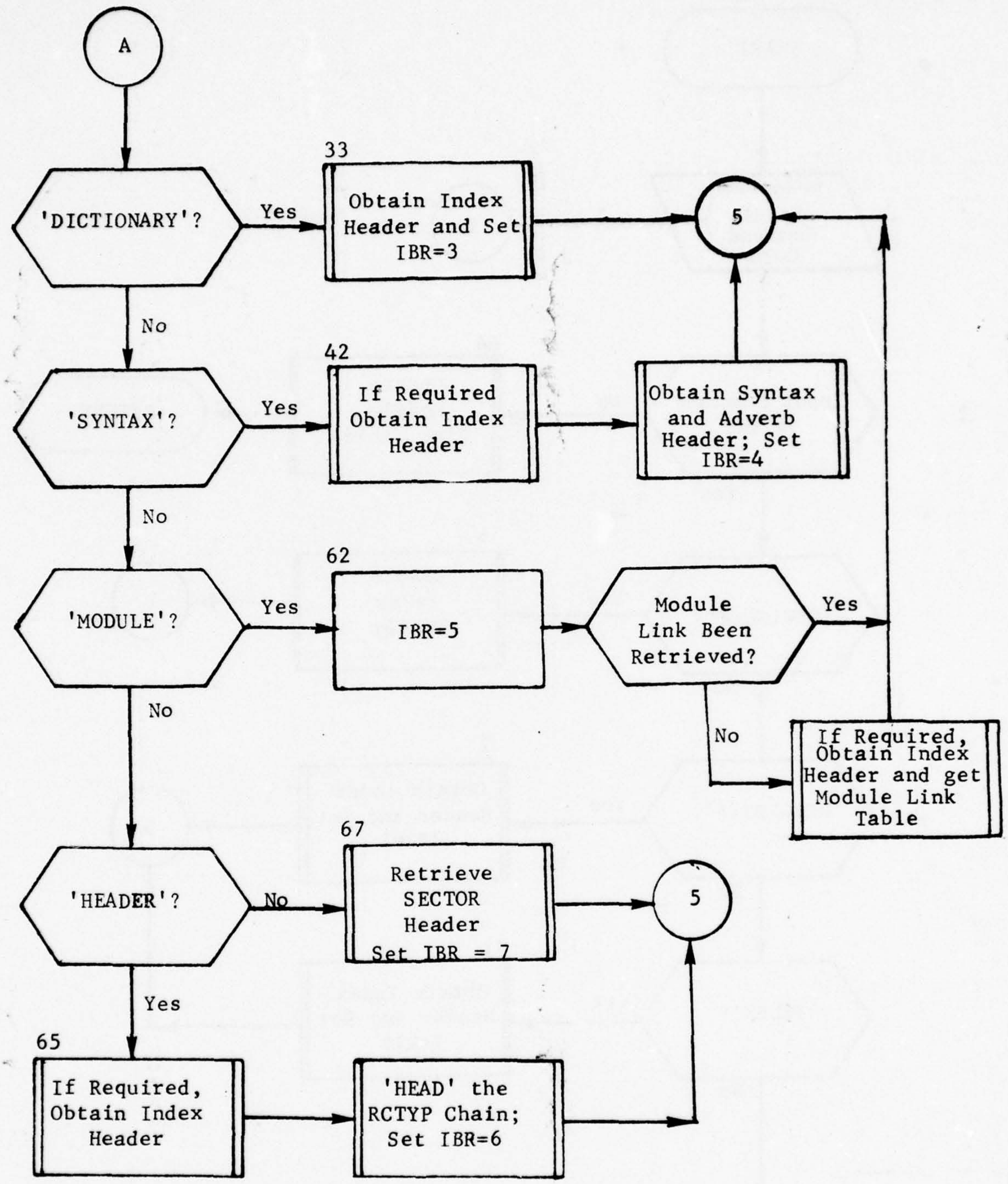



Figure 21. (Part 2 of 26)

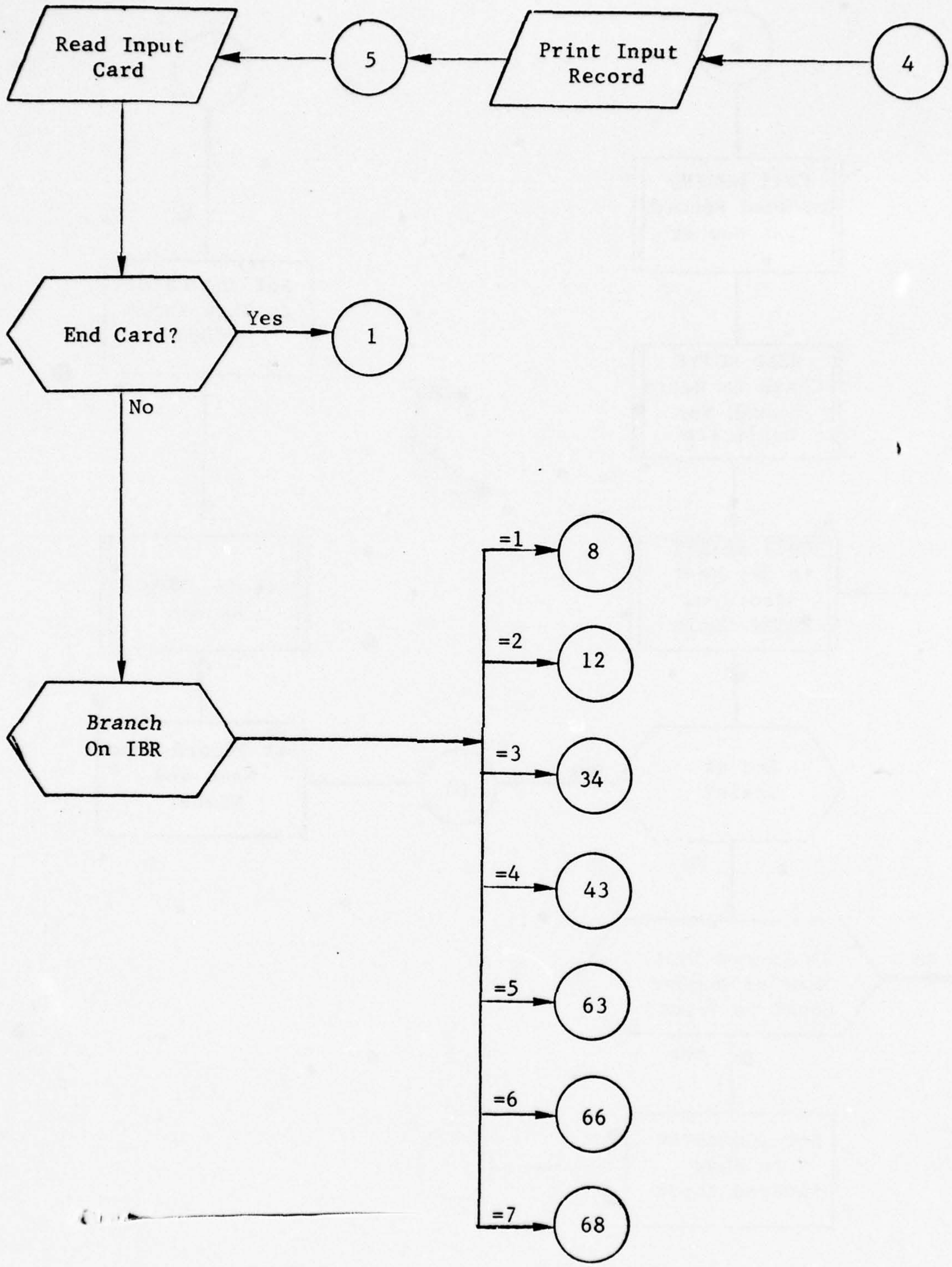



Figure 21. (Part 3 of 26)

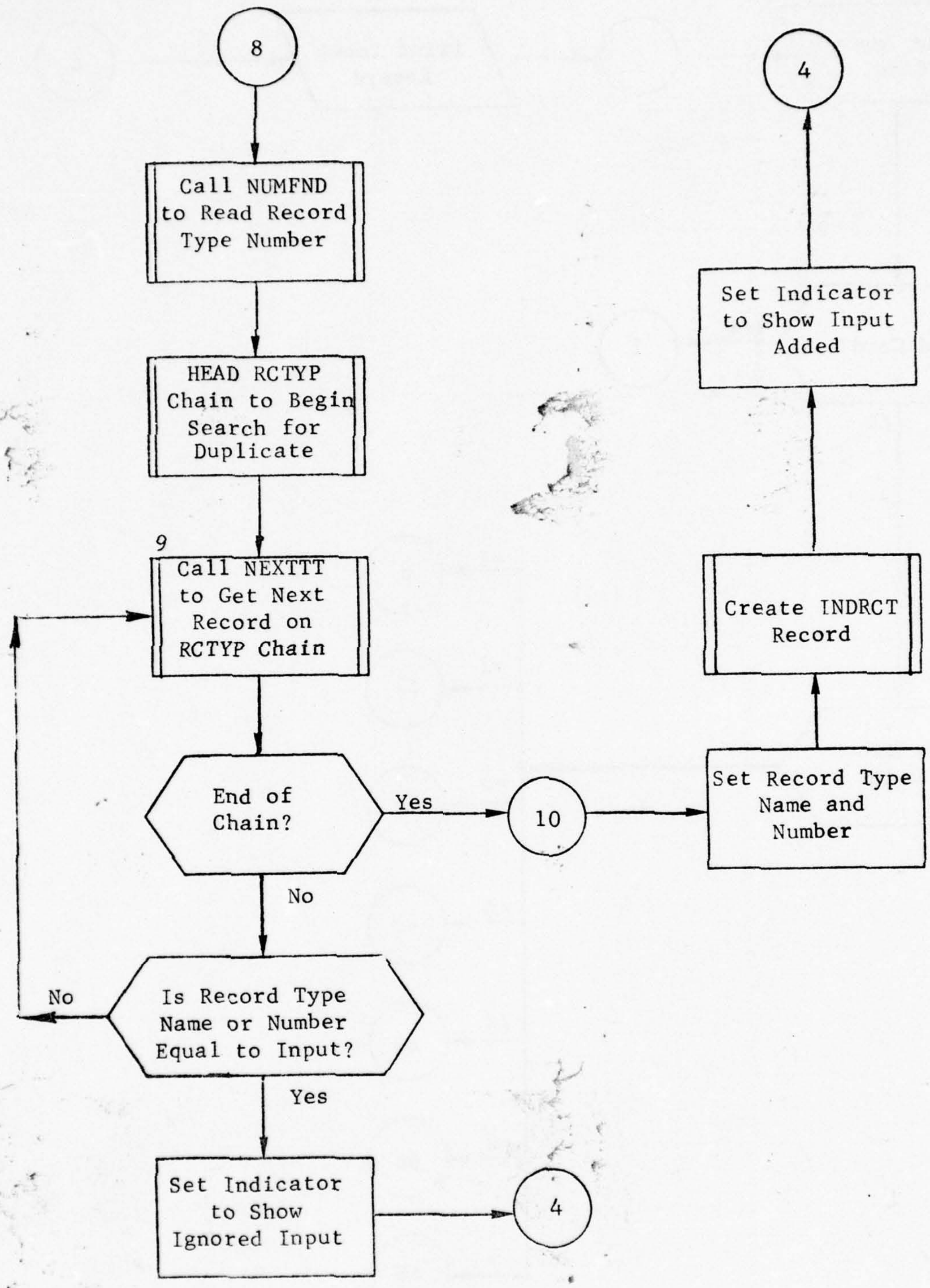



Figure 21. (Part 4 of 26)

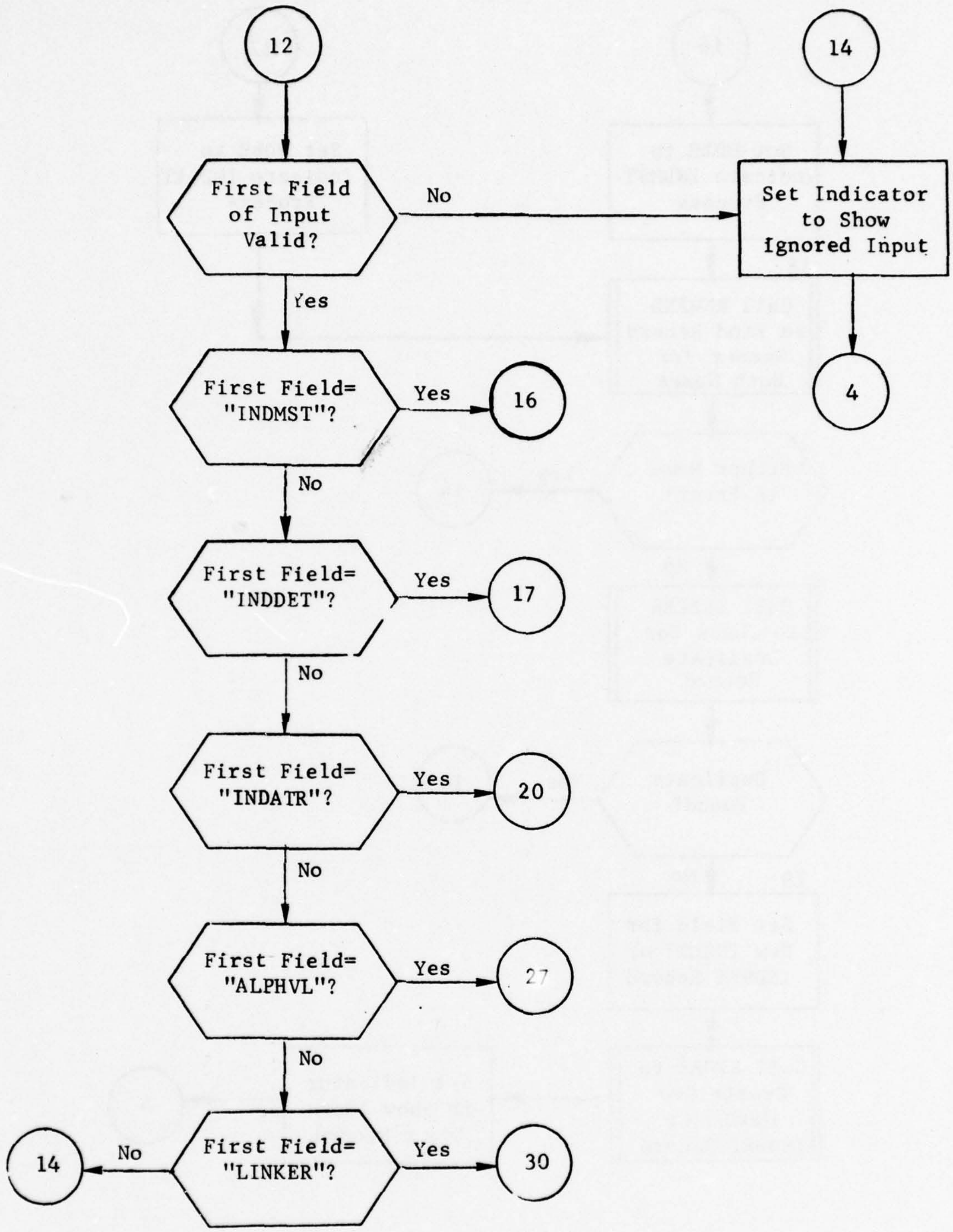



Figure 21. (Part 5 of 26)

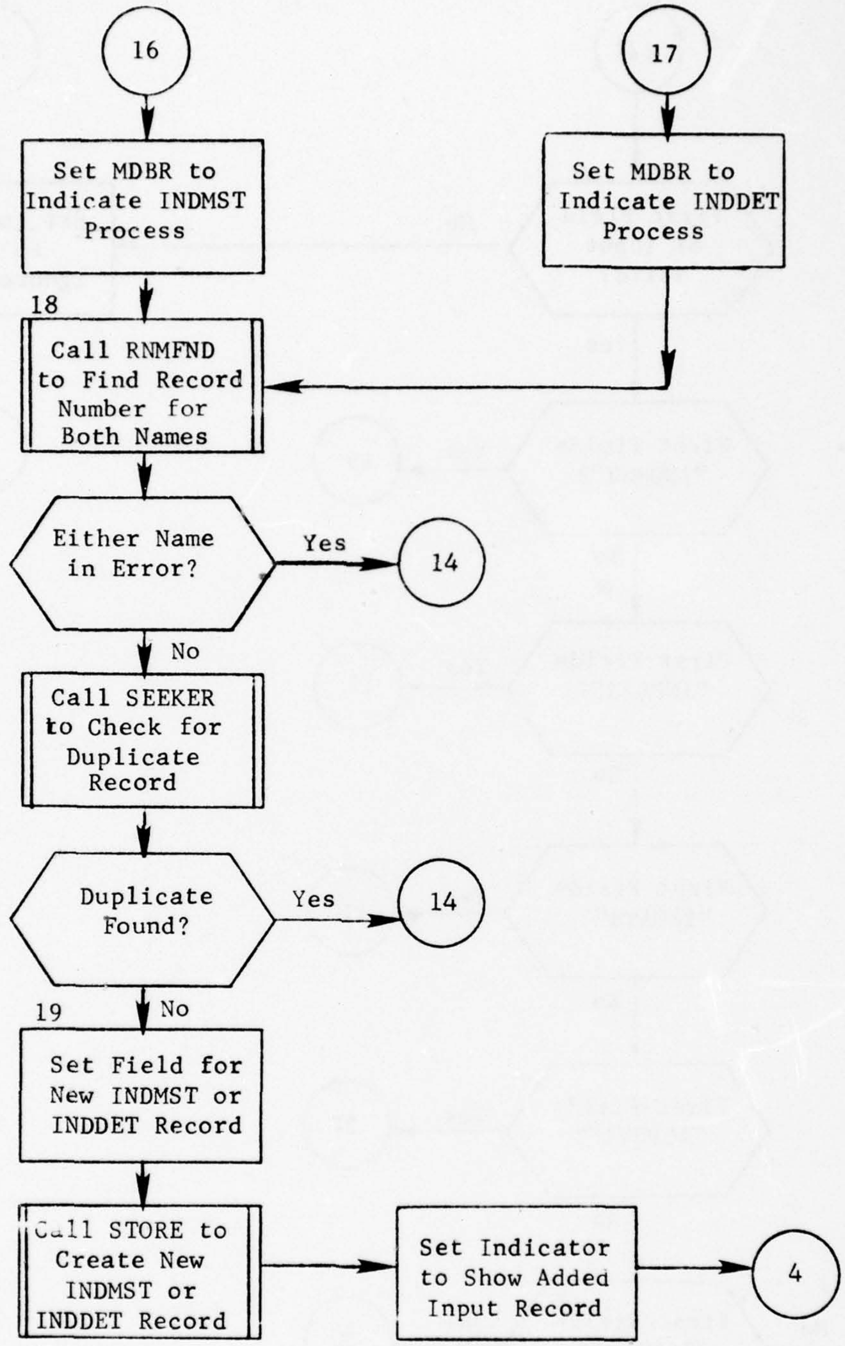



Figure 21. (Part 6 of 26)

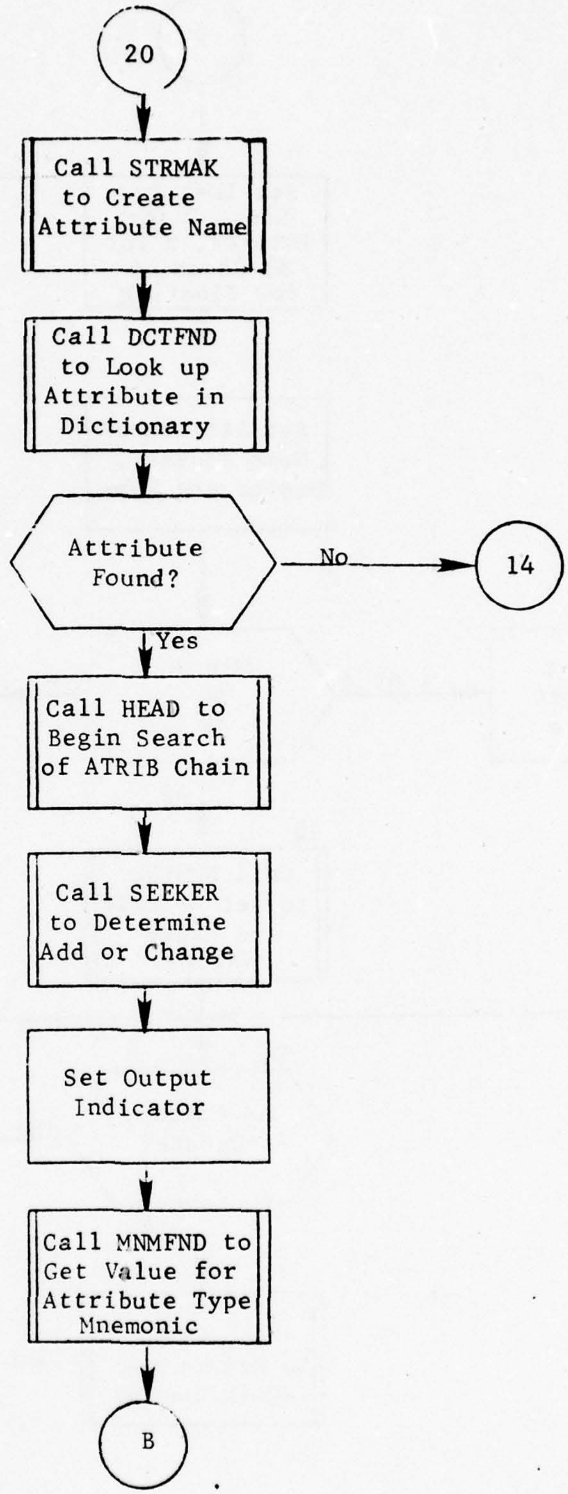



Figure 21. (Part 7 of 26)

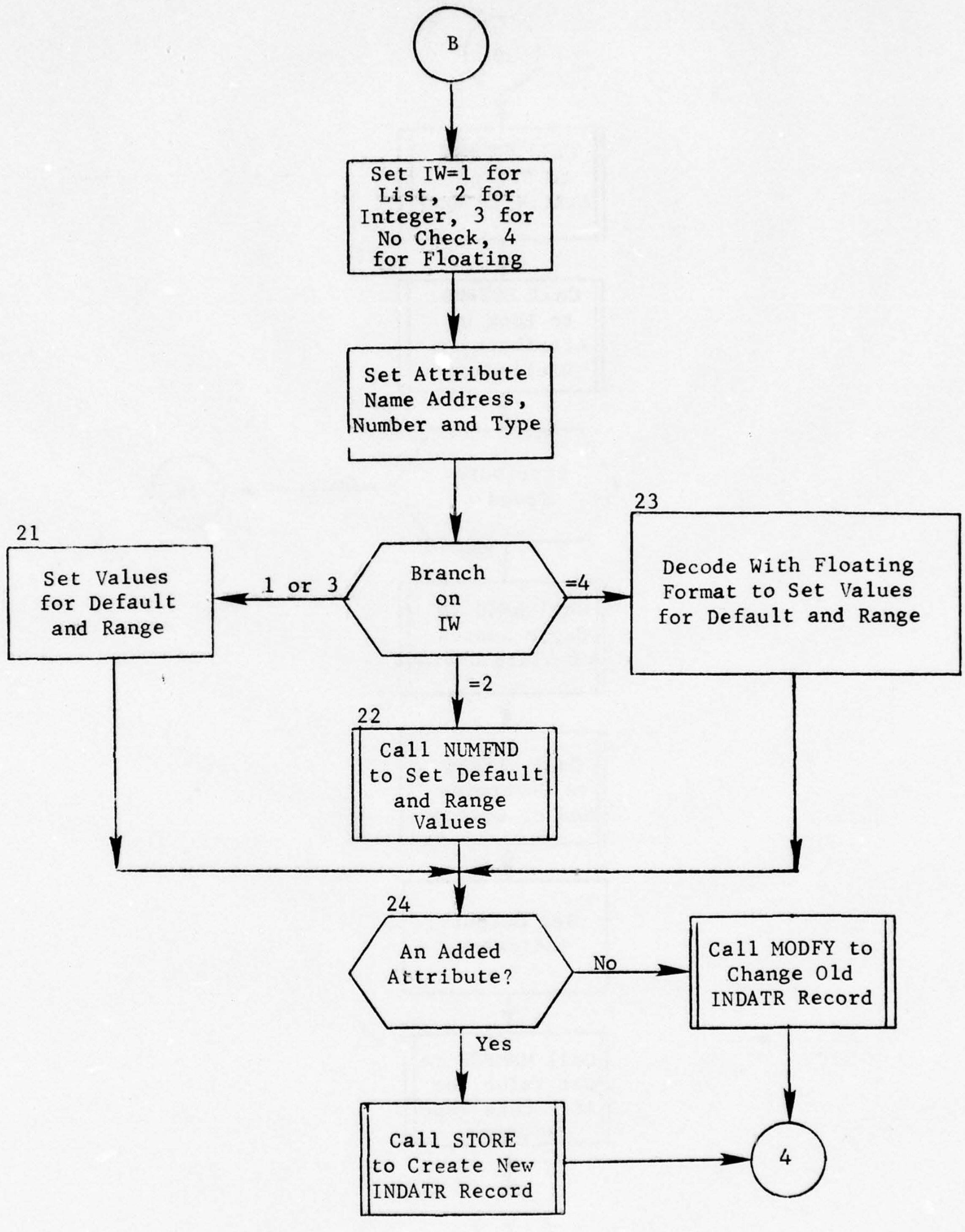



Figure 21. (Part 8 of 26)

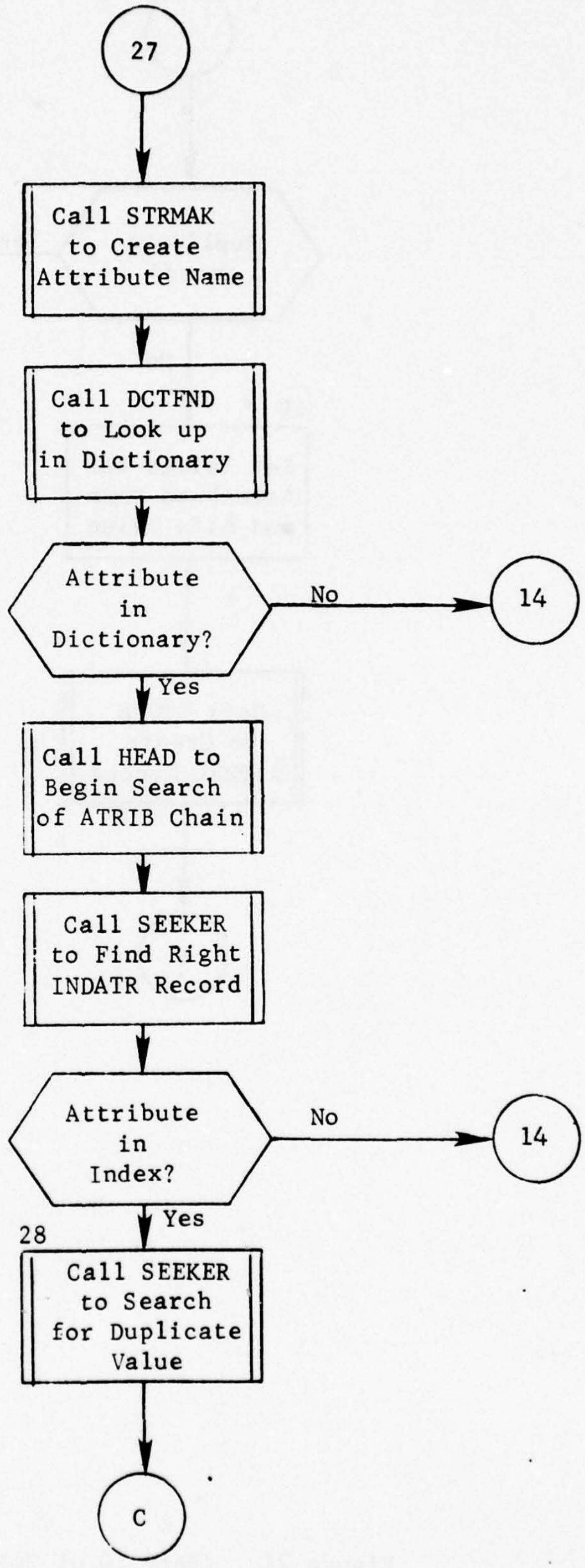



Figure 21. (Part 9 of 26)

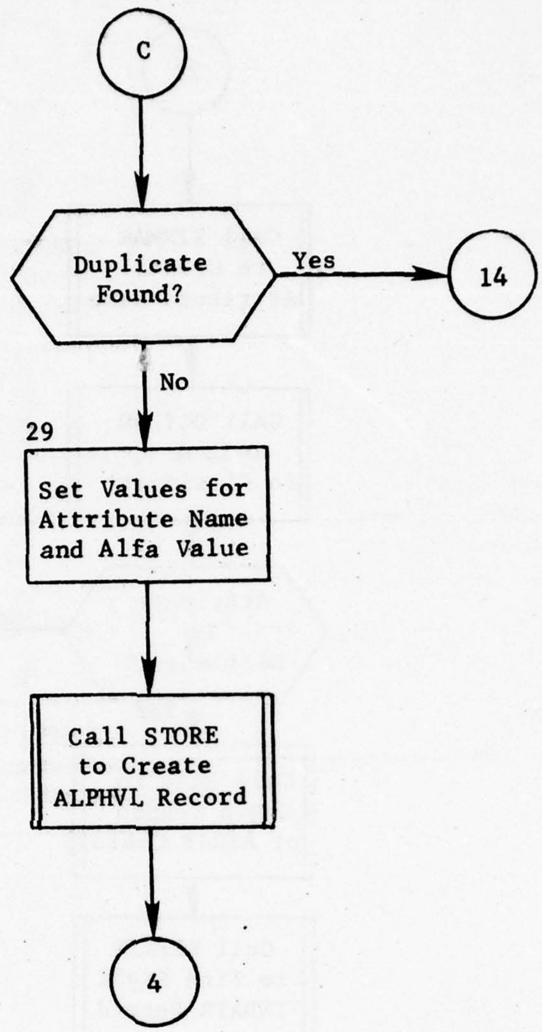



Figure 21. (Part 10 of 26)

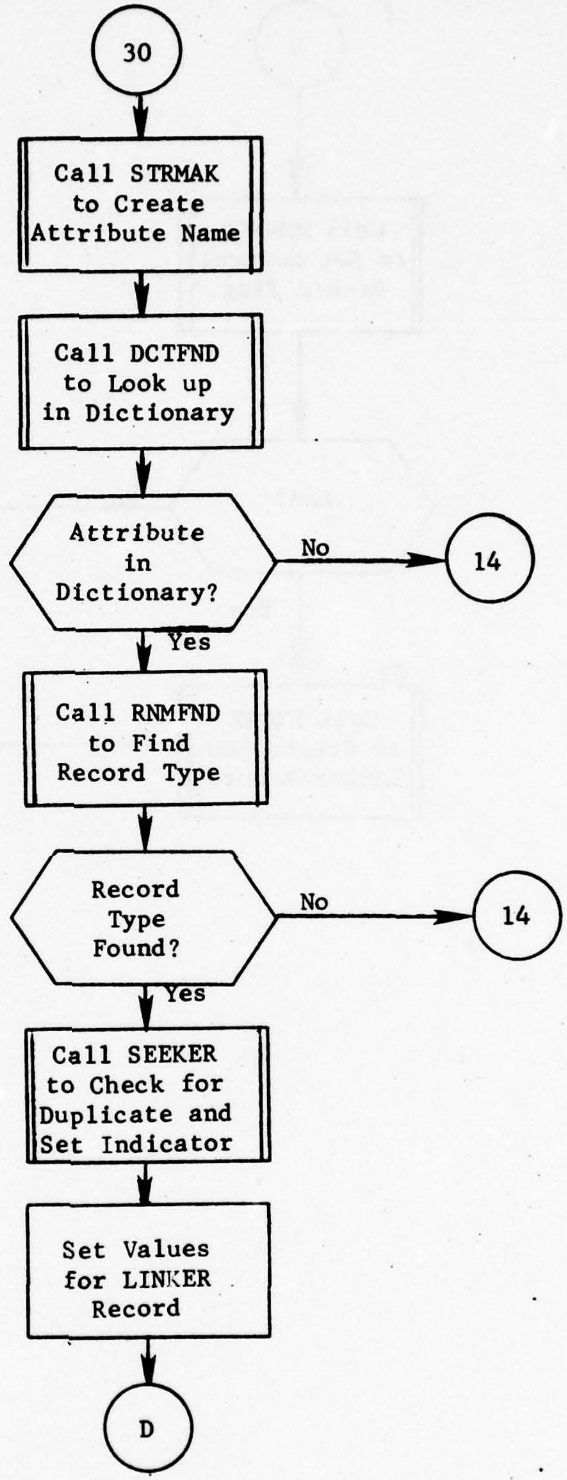



Figure 21. (Part 11 of 26)

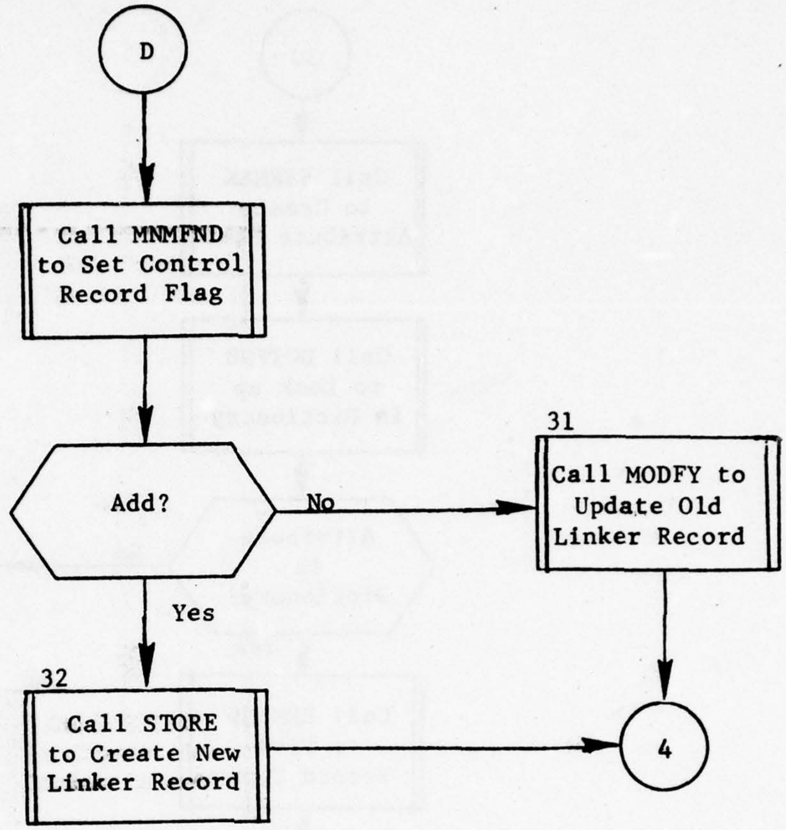



Figure 21. (Part 12 of 26)

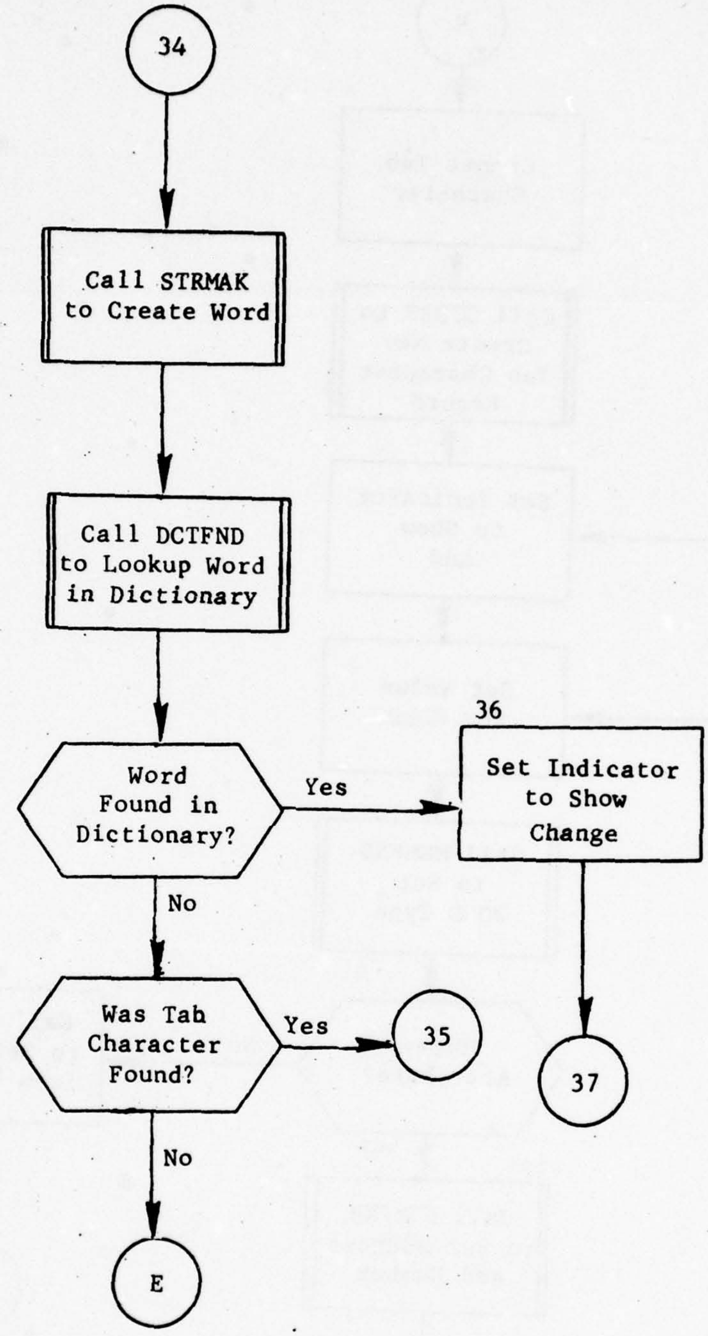



Figure 21. (Part 13 of 26)

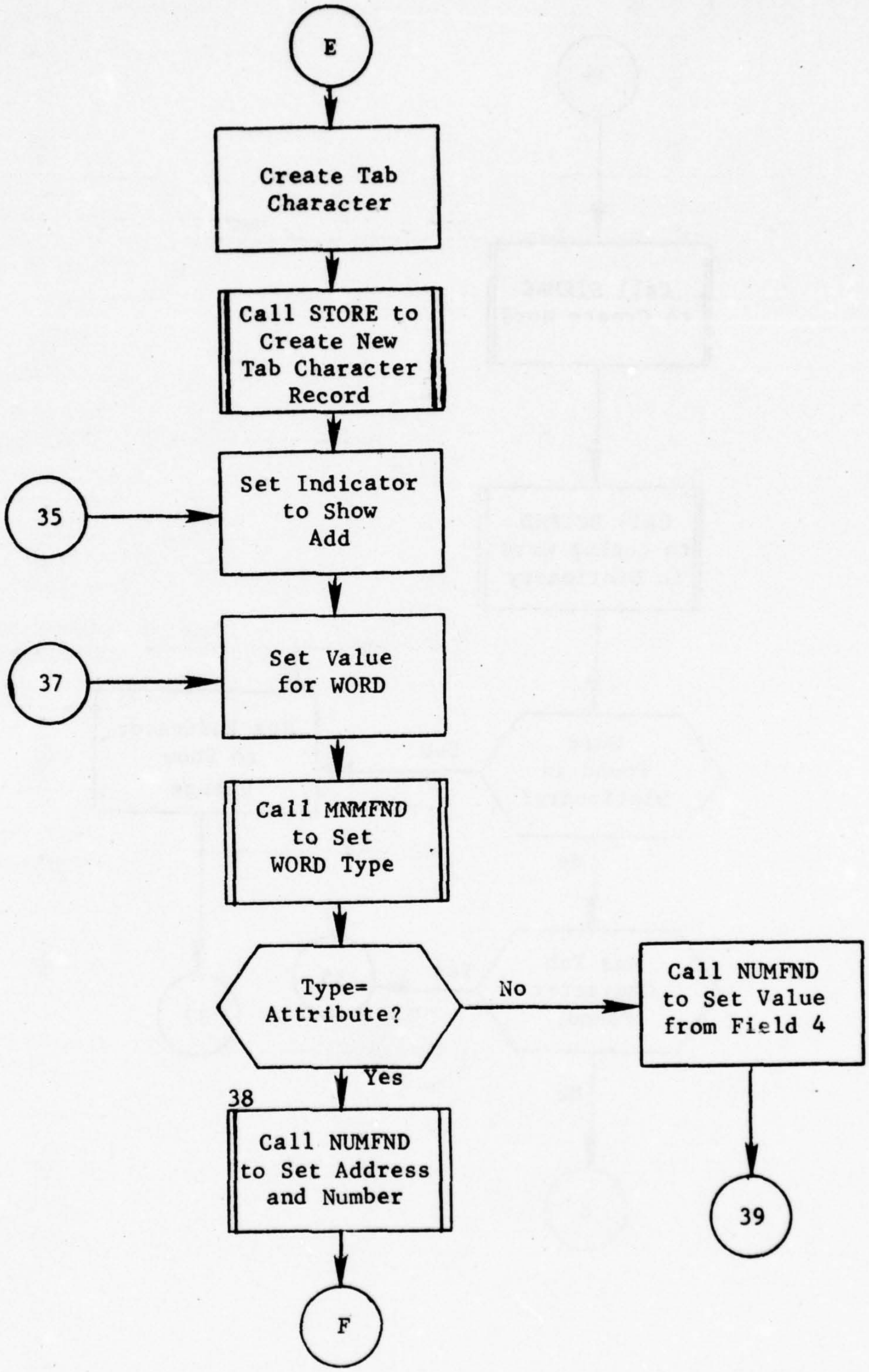



Figure 21. (Part 14 of 26)

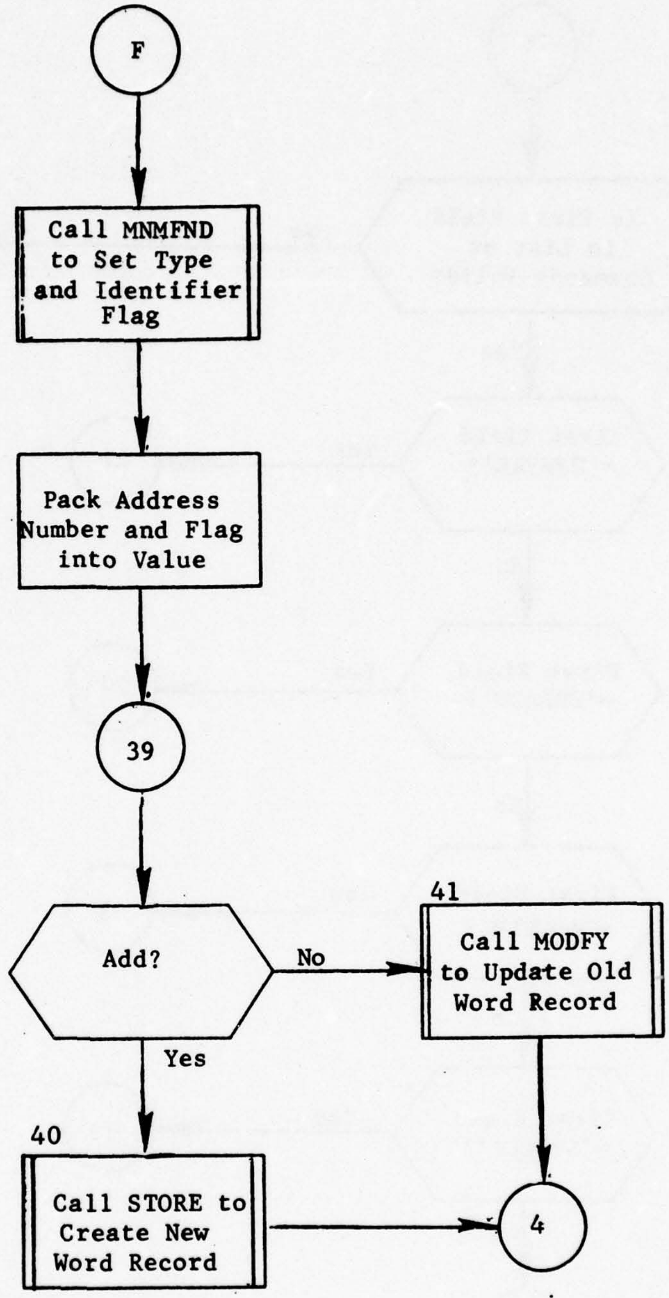



Figure 21. (Part 15 of 26)

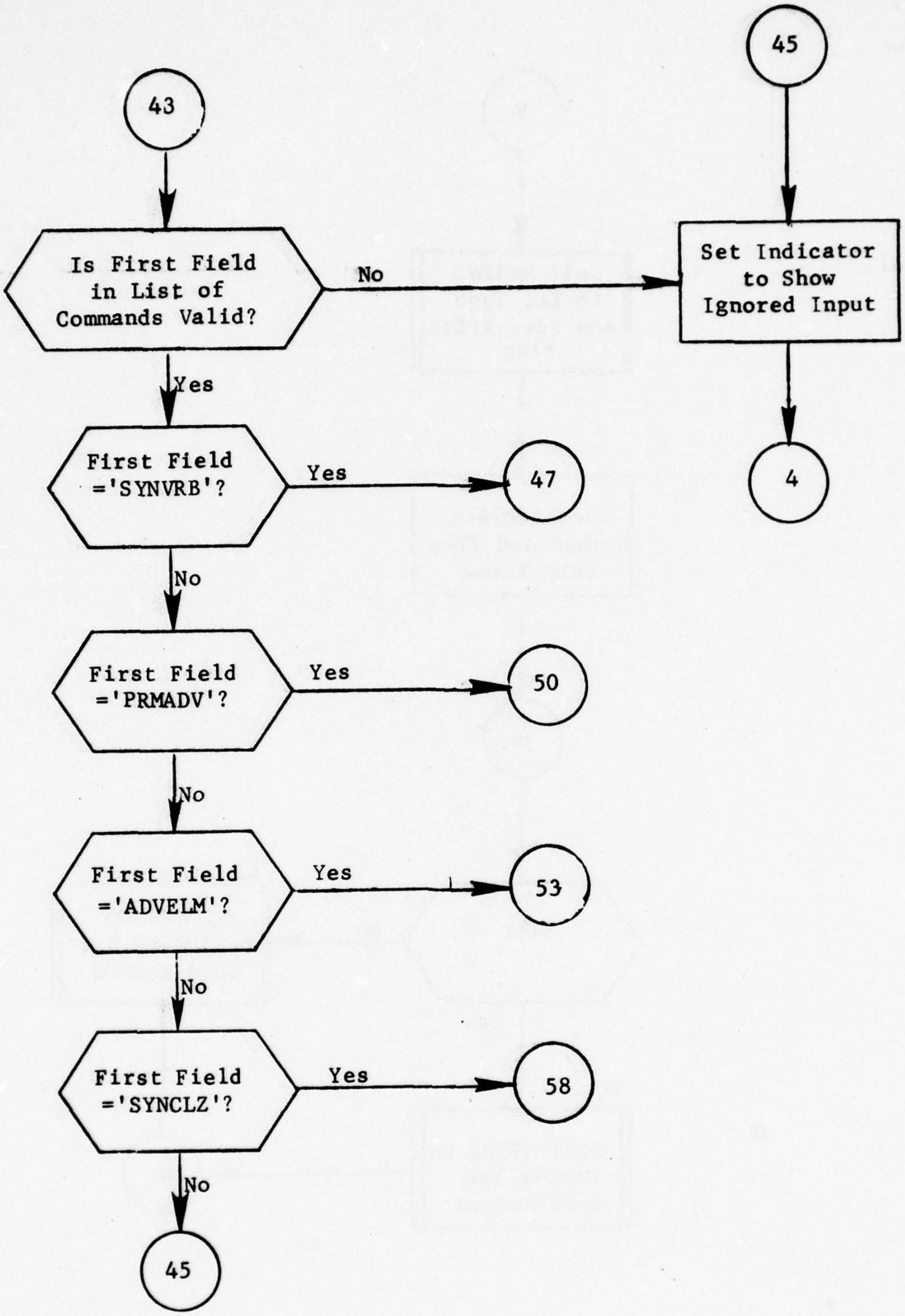



Figure 21. (Part 16 of 26)

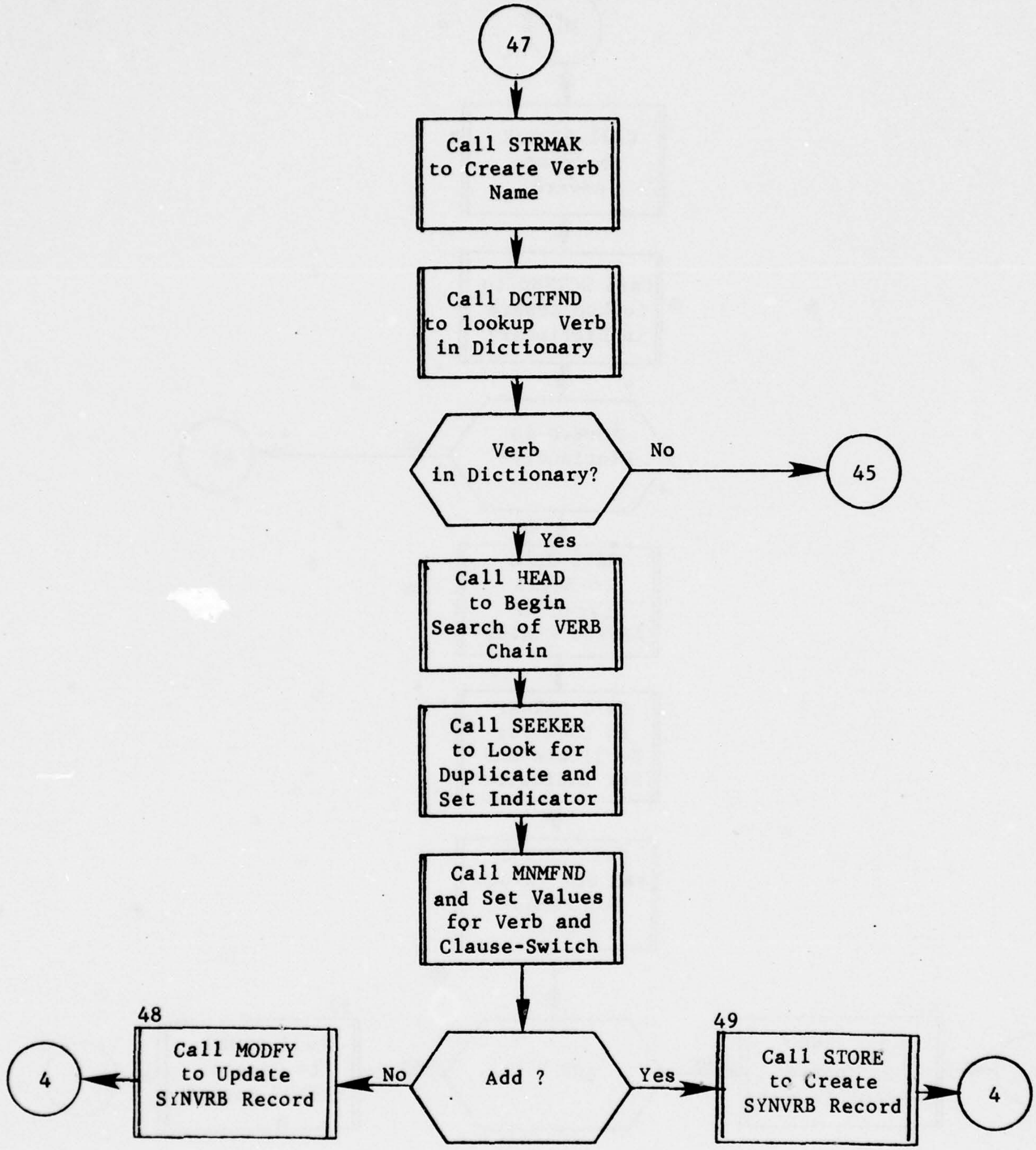



Figure 21. (Part 17 of 26)

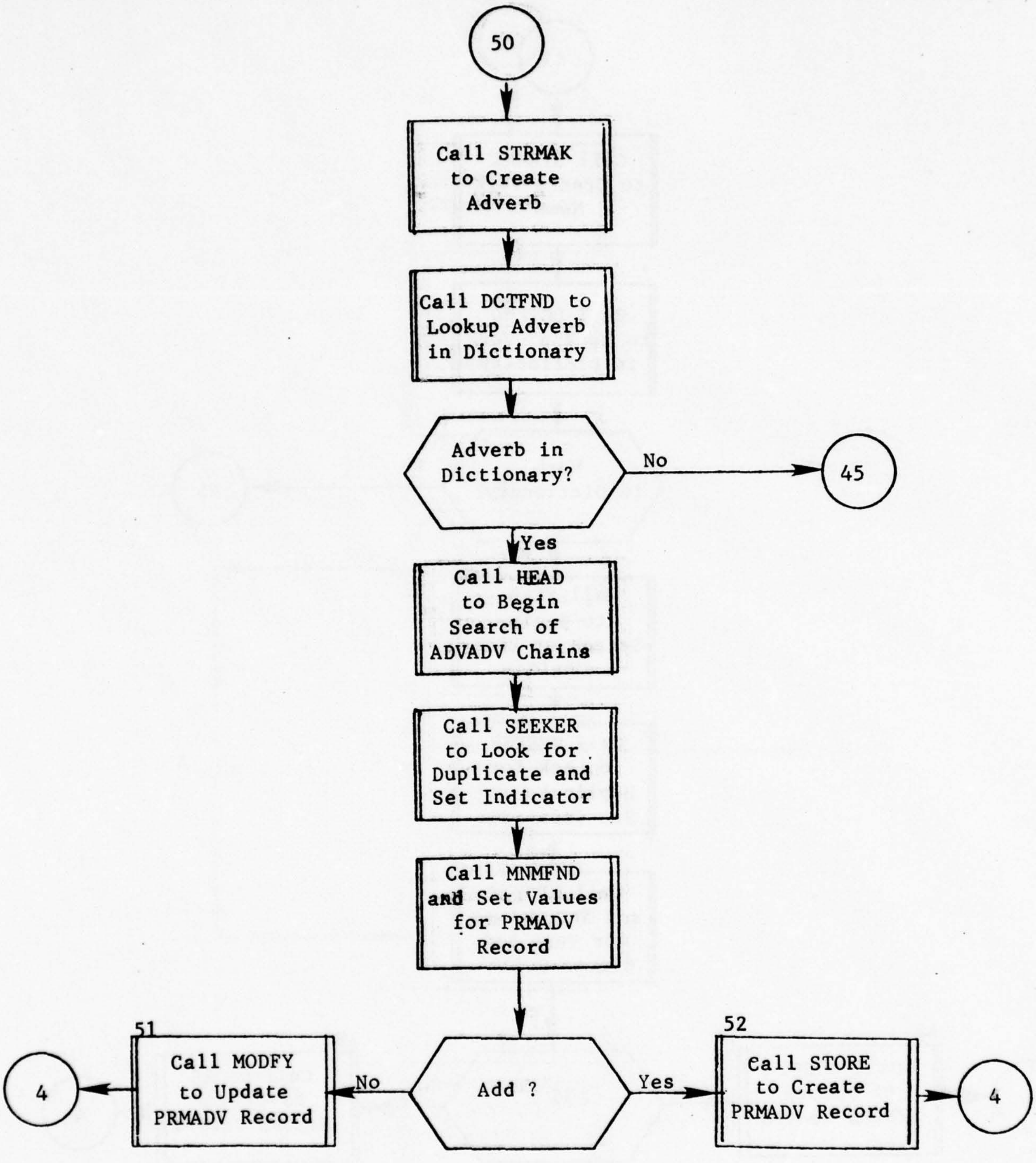



Figure 21. (Part 18 of 26)

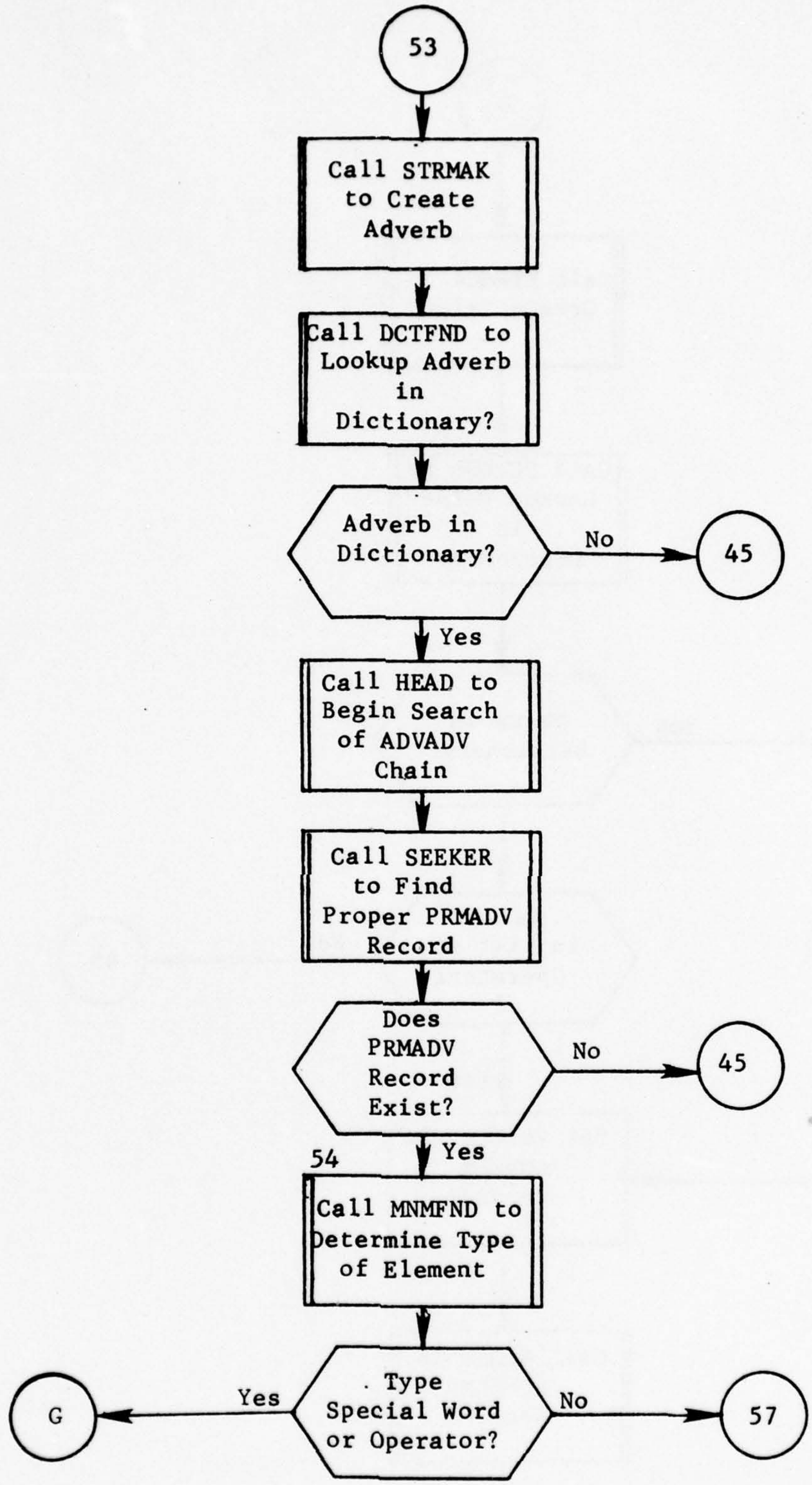



Figure 21. (Part 19 of 26)

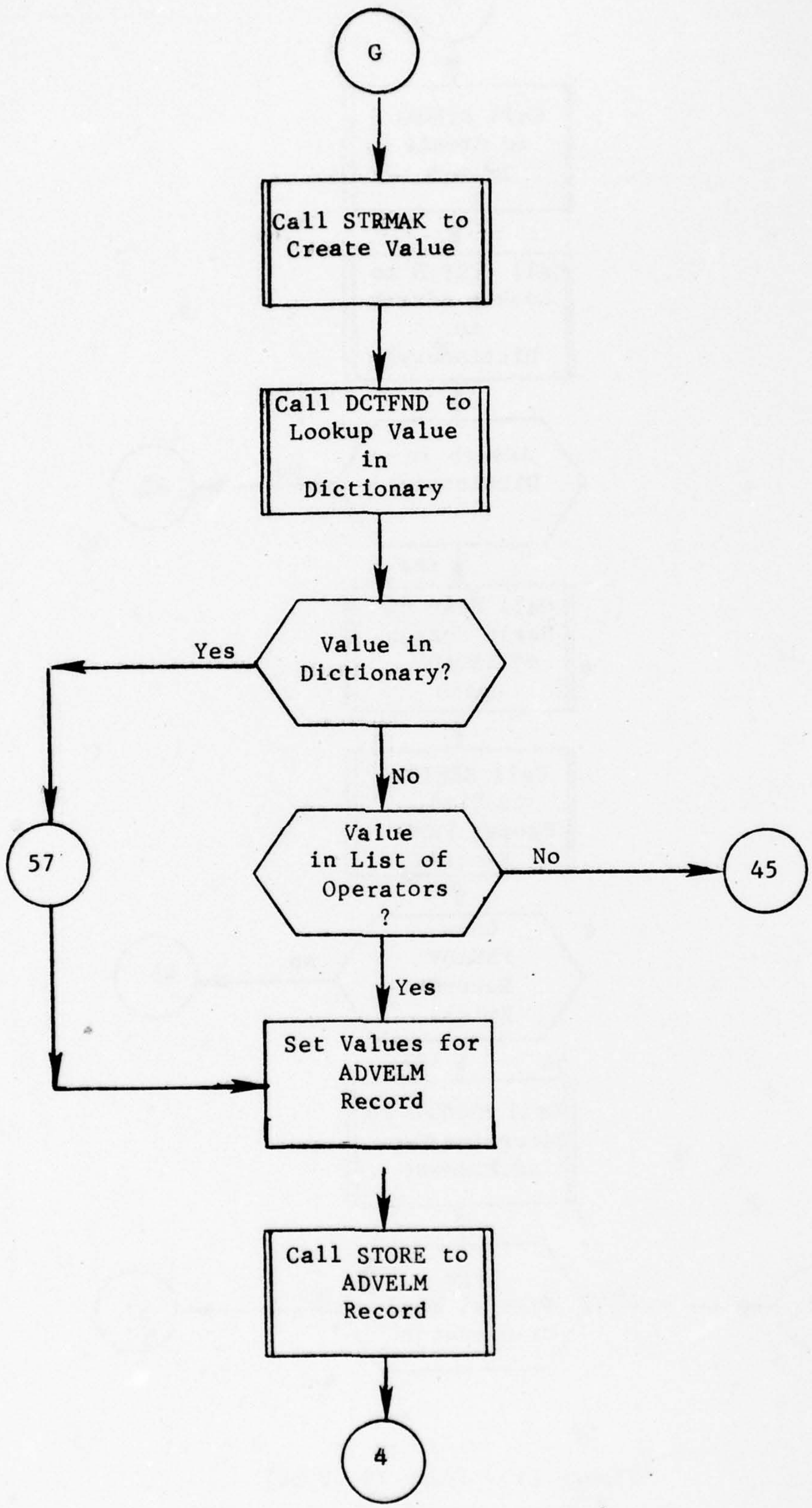



Figure 21. (Part 20 of 26)

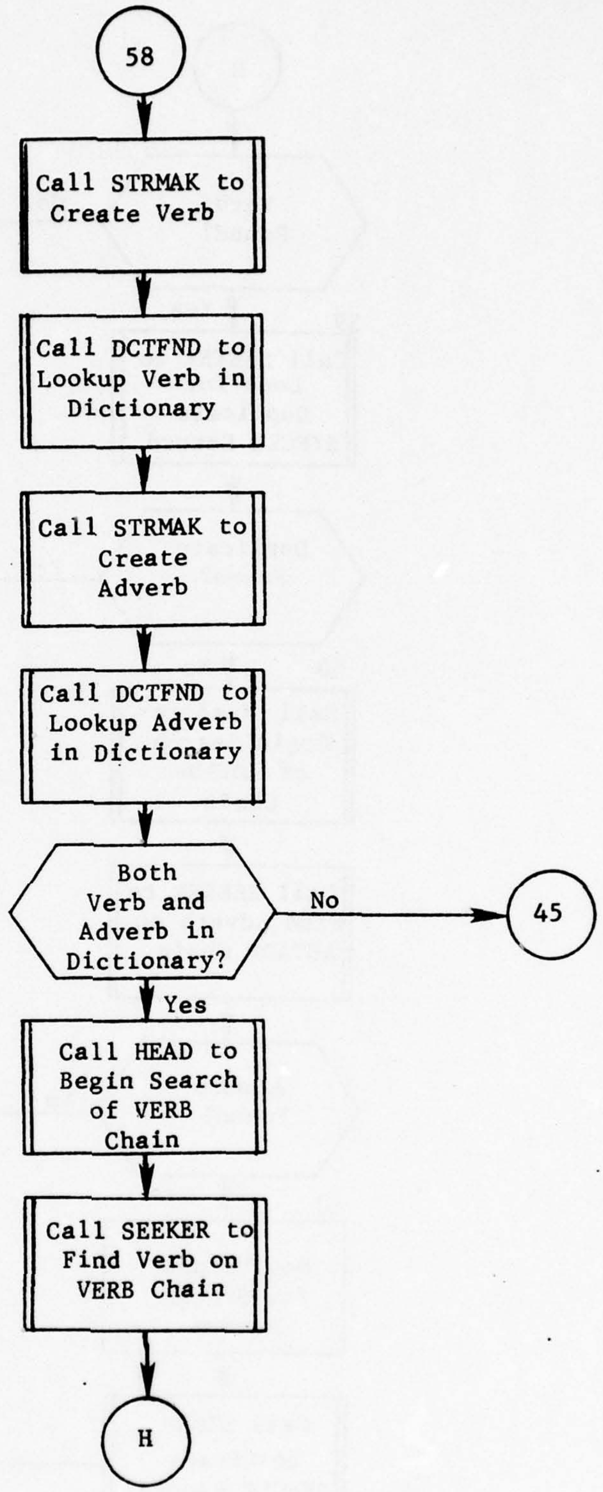



Figure 21. (Part 21 of 26)

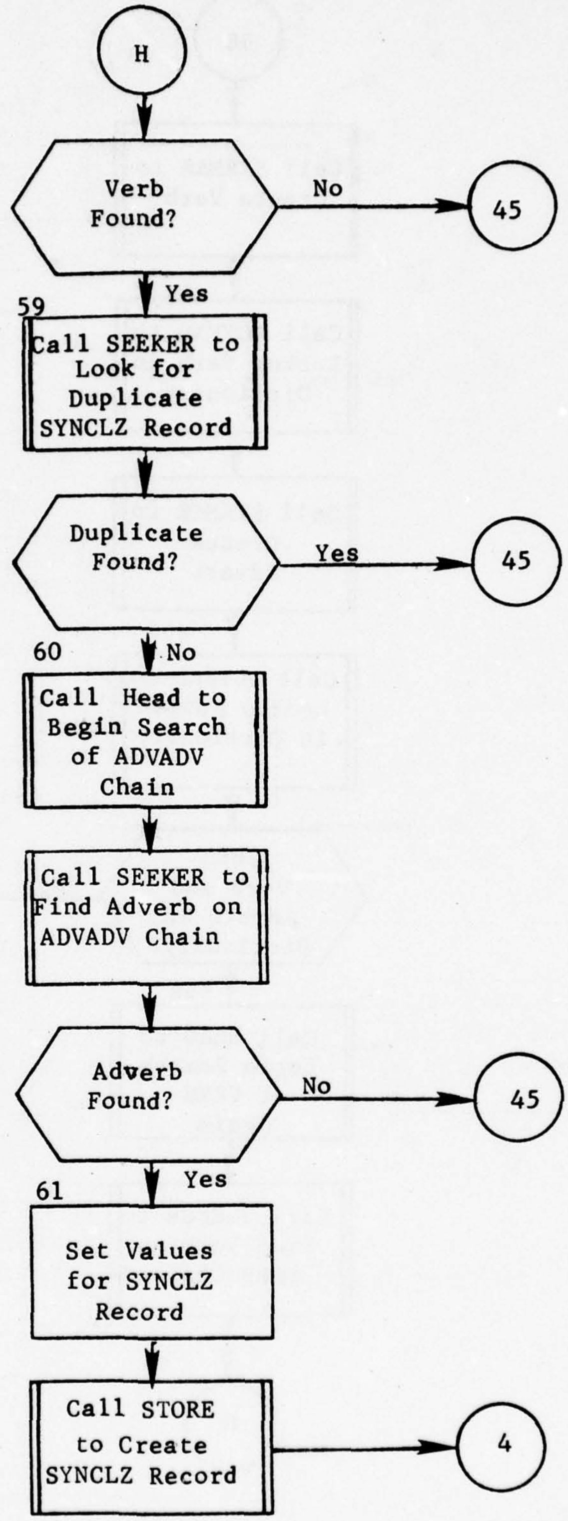



Figure 21. (Part 22 of 26)

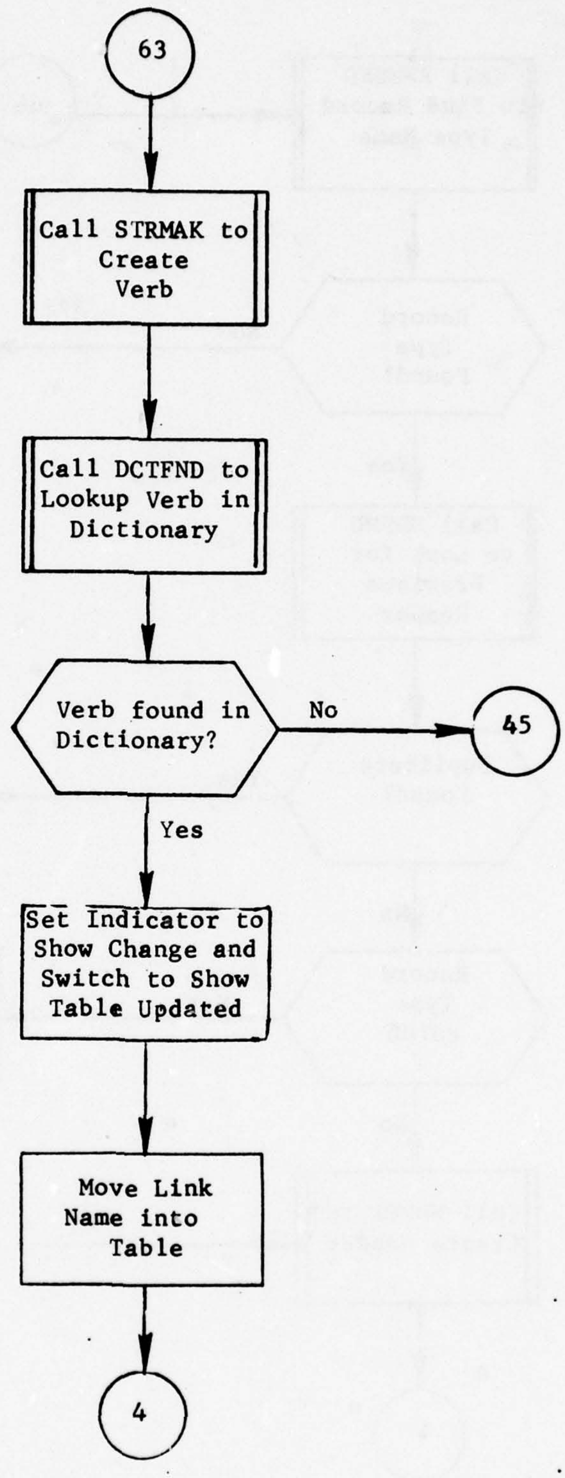



Figure 21. (Part 23 of 26)

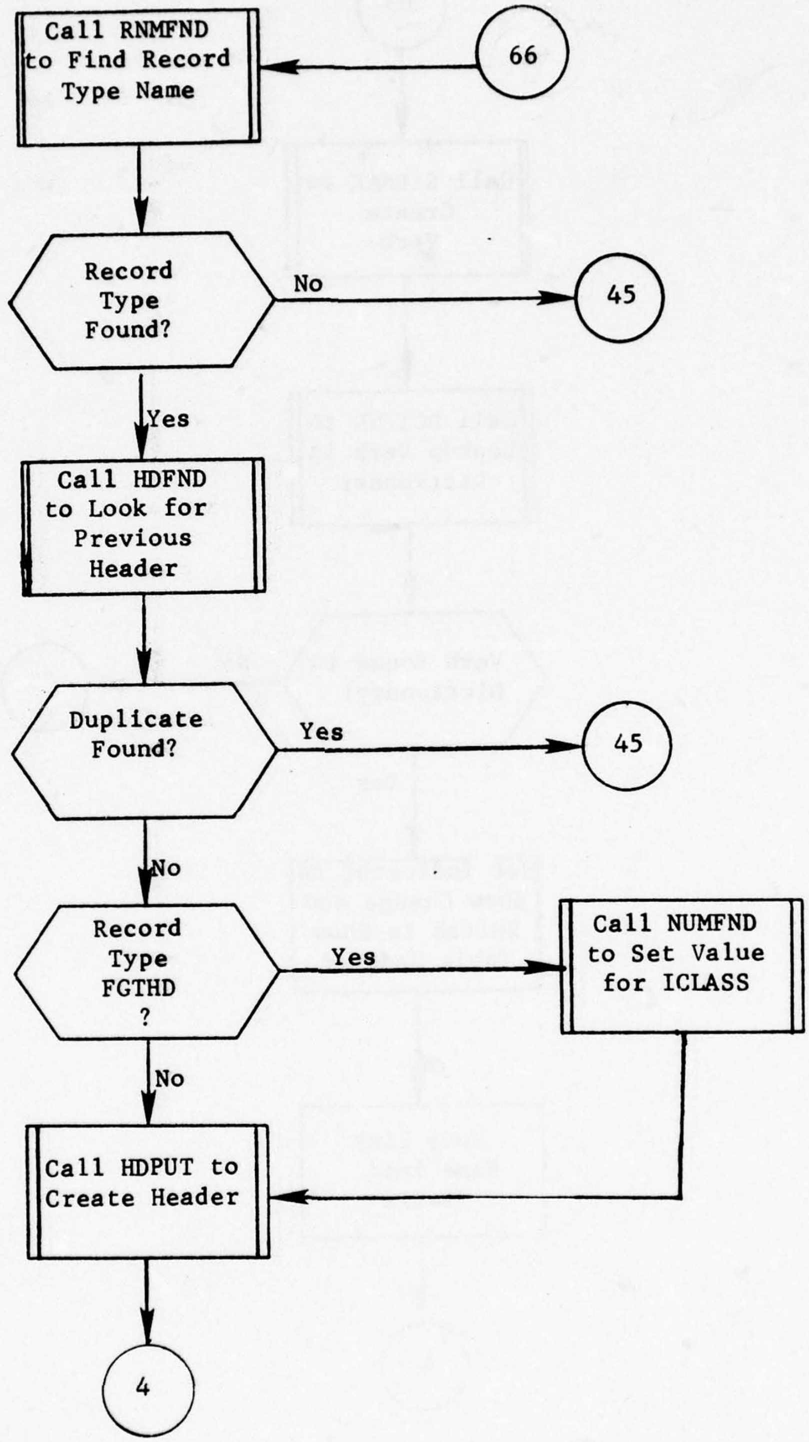



Figure 21. (Part 24 of 26)

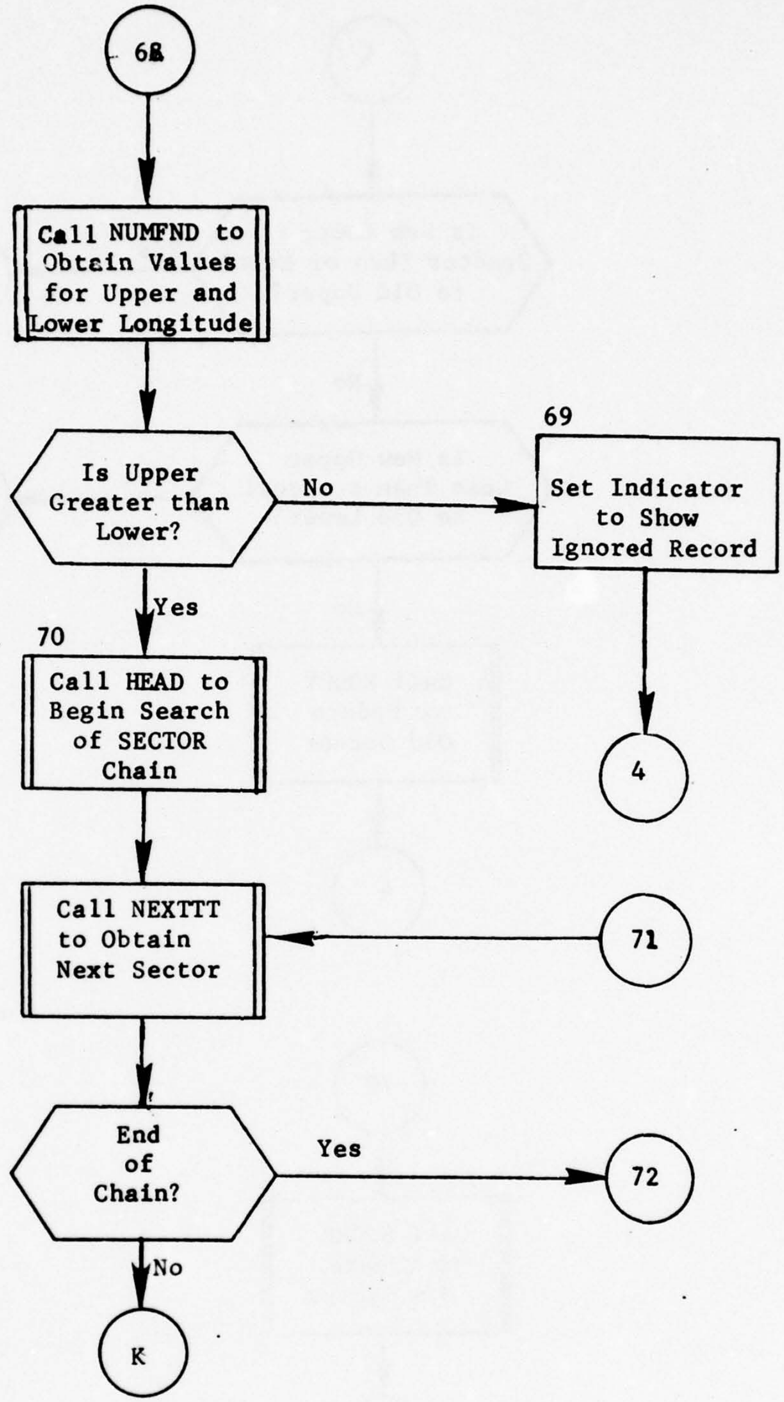



Figure 21. (Part 25 of 26)

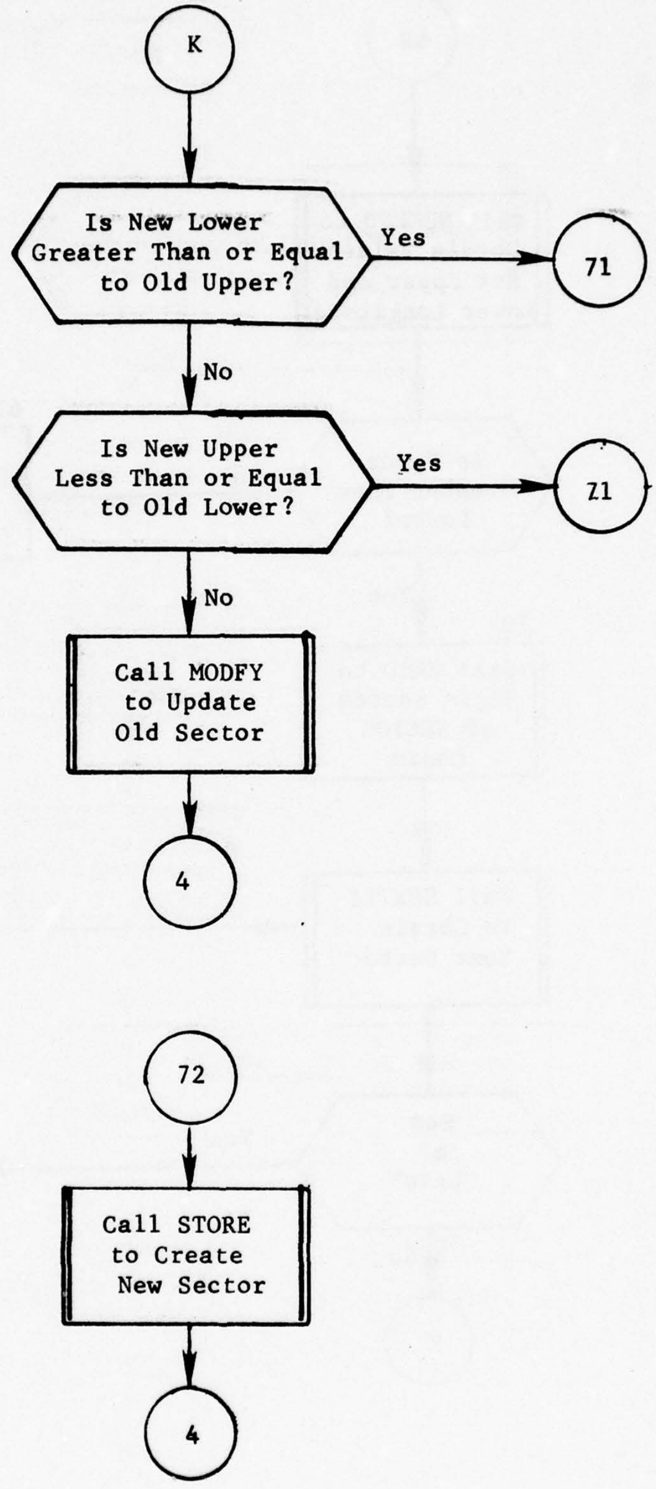



Figure 21. (Part 26 of 26)

### 3.8.1 Subroutine DCTFND

| | |
|---|---|
| PURPOSE: | Check for dictionary match |
| ENTRY POINTS: | DCTFND |
| FORMAL PARAMETERS: | STRING: Character string to be searched for<br>TXPE: Type of string expected<br>NUMBER: Identifying number returned<br>ADRESS: Address of attribute returned |
| COMMON BLOCKS: | C10, C15, C30 |
| SUBROUTINES CALLED: | HDFND, NEXTTT, RETRV |
| CALLED BY: | BOOT |

Method:

First the HDSAVE variable is checked. If blank HDFND is called to set the value of the reference code of the dictionary header into HDSAVE. The dictionary header is retrieved. ICOMP is set to the first two characters of the input string to be used as a tab character. The TAB chain is now searched to find a match for ICOMP. If not found, NUMBER is set to zero and the subroutine returns.

If a match is found for ICOMP, the WORD chain is searched for a match. If none is found, the subroutine returns with NUMBER=0. If a match is found the type of the word found is compared to TXPE. If no match, NUMBER is set to zero. If a match, NUMBER is set from the dictionary and, if the type is attribute, ADRESS is also set.

Subroutine DCTFND is illustrated in figure 22.

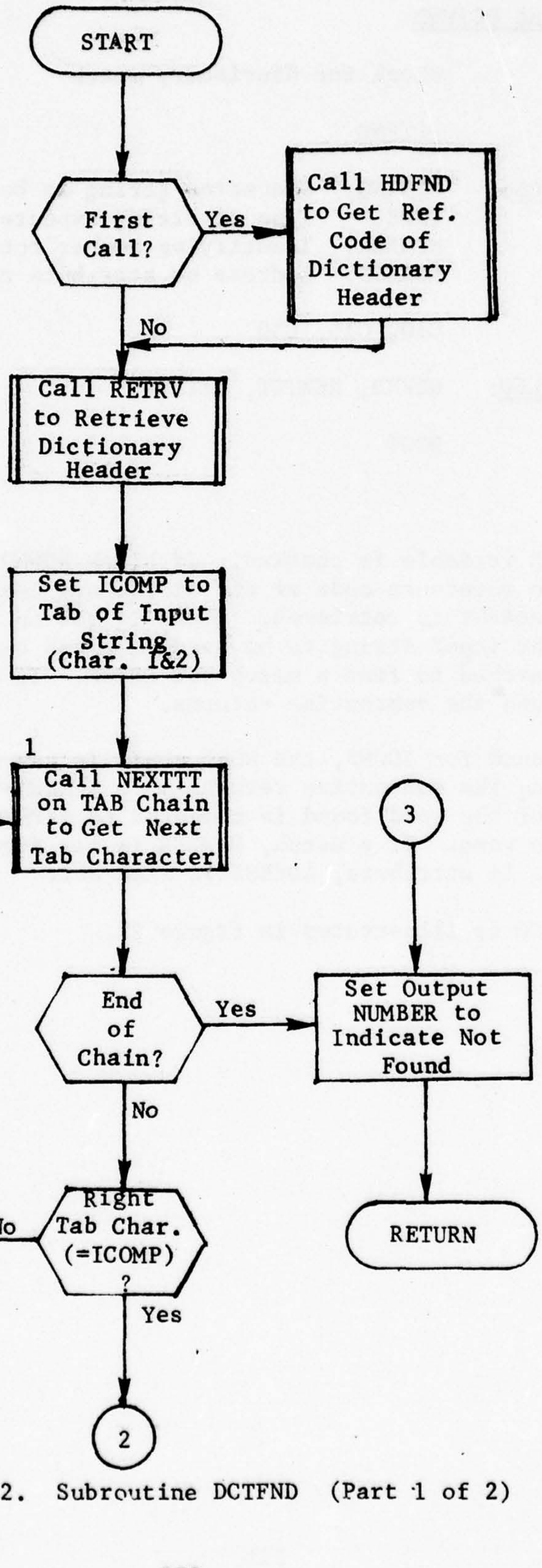



Figure 22. Subroutine DCTFND (Part 1 of 2)

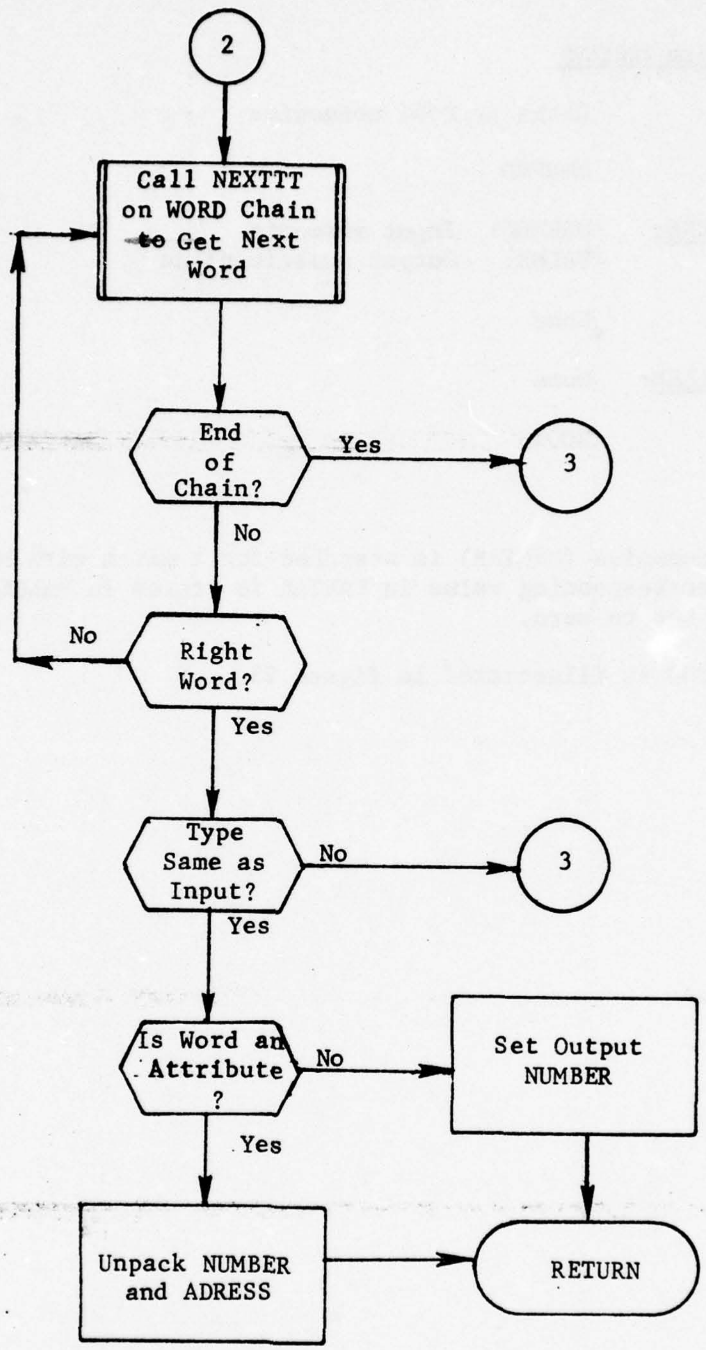



Figure 22. (Part 2 of 2)

### 3.3.2 Subroutine MNMFND

| | |
|---|---|
| PURPOSE: | Looks up BOOT mnemonics |
| ENTRY POINTS: | MNMFND |
| FORMAL PARAMETERS: | MNEMON: Input mnemonic<br>VALUE: Output numeric value |
| COMMON BLOCKS: | None |
| SUBROUTINES CALLED: | None |
| CALLED BY: | BOOT |

Method:

The table of mnemonics (MNMTAB) is searched for a match with MNEMON. If found, the corresponding value in VALTAB is stored in VALUE. If not found VALUE is set to zero.

Subroutine MNMFND is illustrated in figure 23.

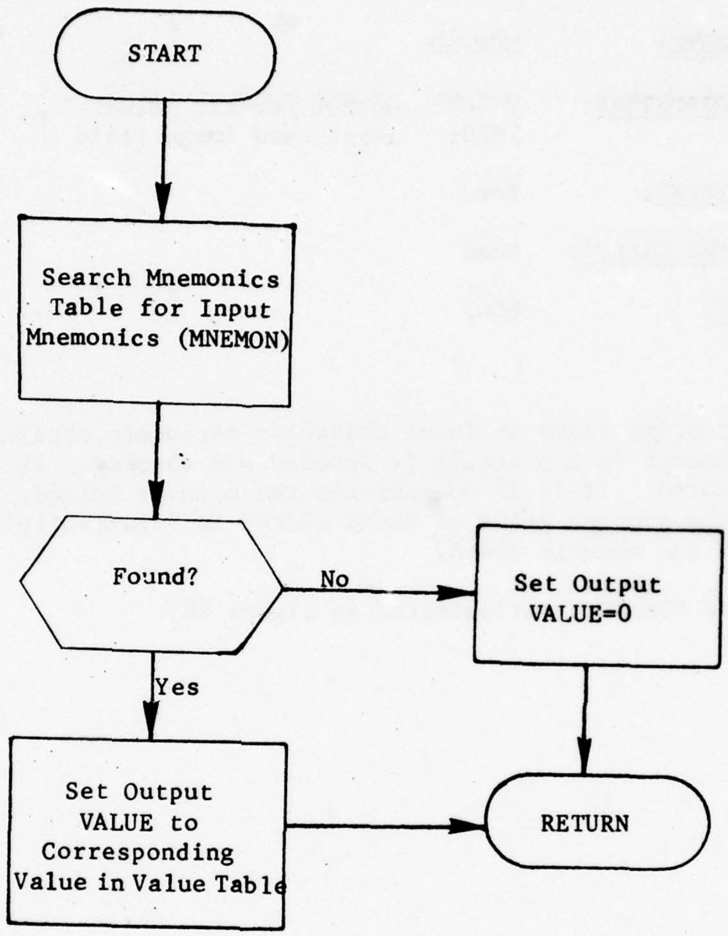



Figure 23. Subroutine MNMFND

### 3.8.3 Subroutine NUMFND

| | |
|---|---|
| *PURPOSE:* | *Finds number in field of input card image for BOOT* |
| ENTRY POINTS: | NUMFND |
| FORMAL PARAMETERS: | VALUE: Output numeric value<br>FELD: Input card image field |
| COMMON BLOCKS: | None |
| SUBROUTINES CALLED: | None |
| CALLED BY: | BOOT |

Method:

This subroutine scans an input character string to obtain a number. Each character in the string is decoded and checked. If it is a blank it is ignored. If it is non-numeric the scan is halted. If it is numeric the current value of VALUE (VALUE is 0 initially) is multiplied by 10 and the numeric added.

Subroutine NUMFND is illustrated in figure 24.

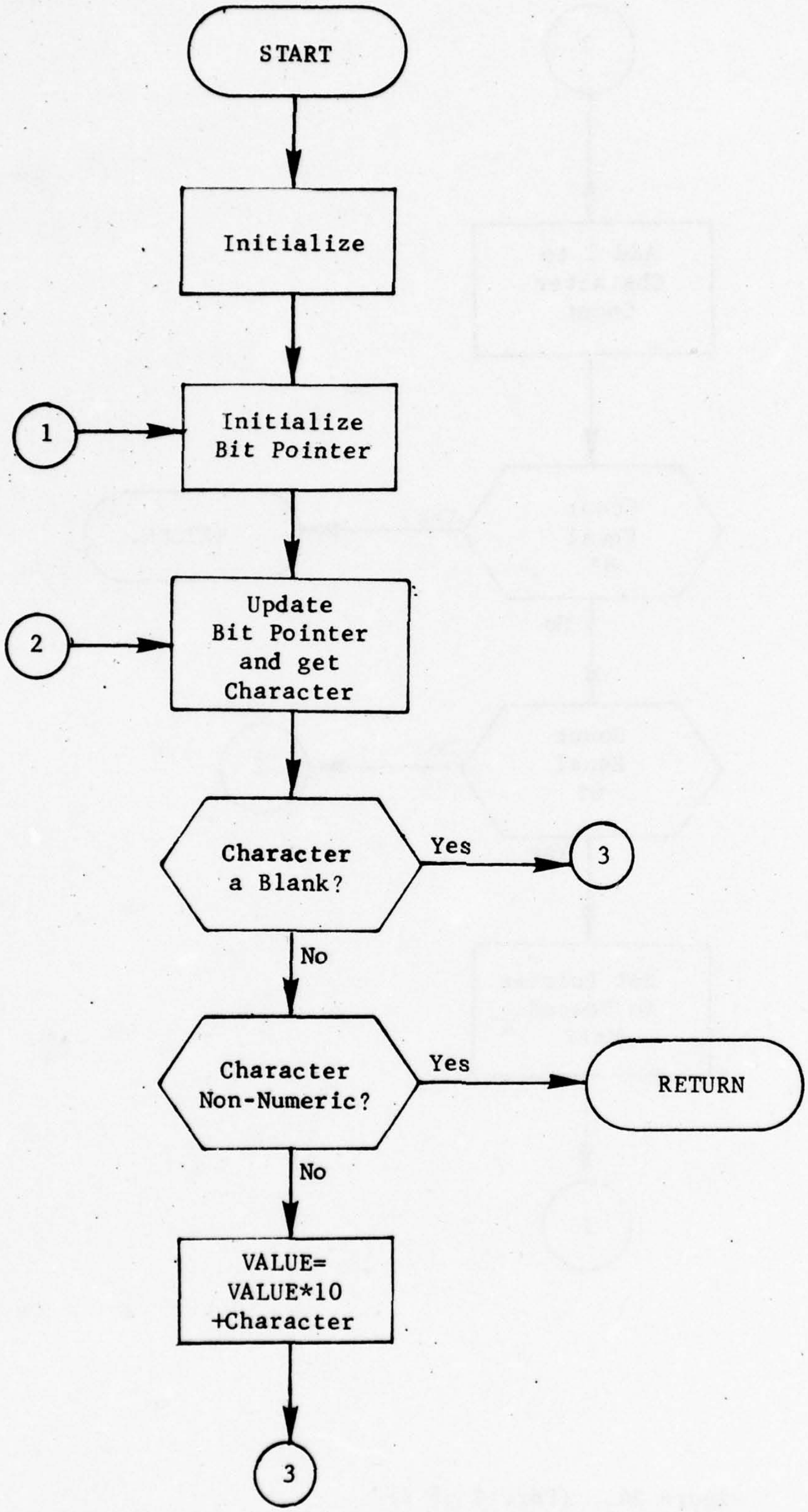



Figure 24. Subroutine NUMFND (Part 1 of 2)

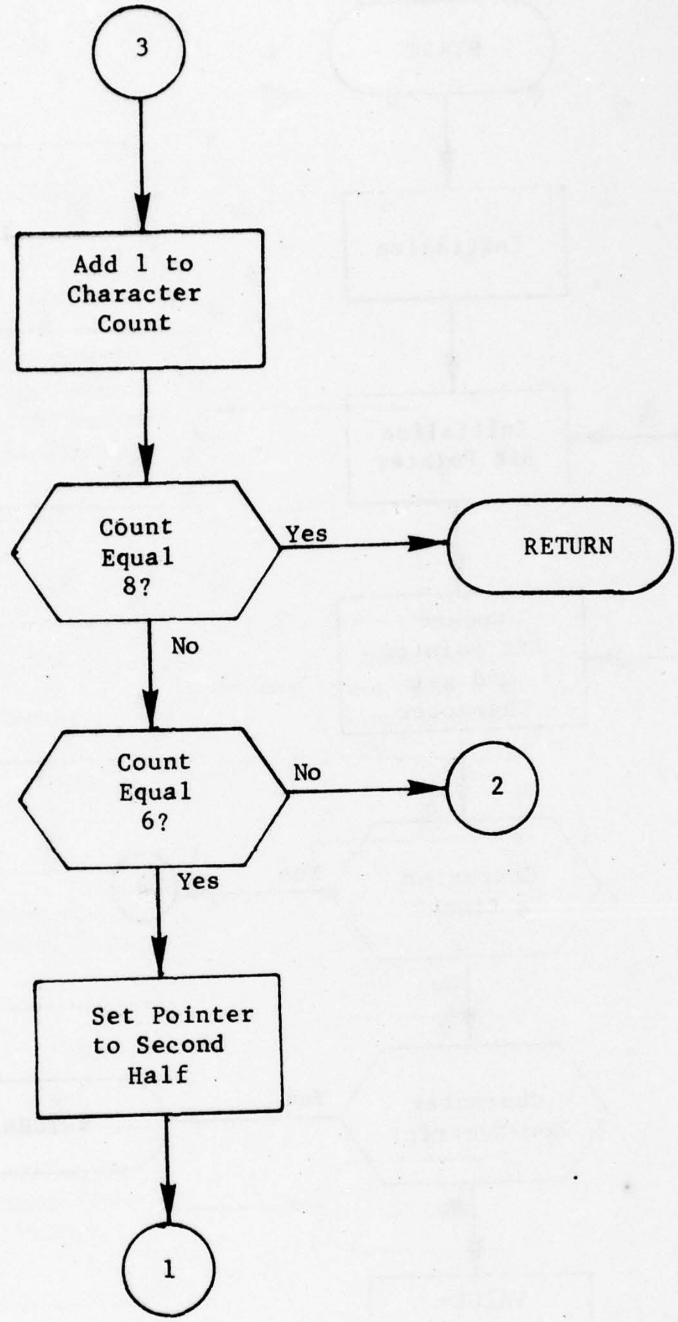



Figure 24. (Part 2 of 2)

### 3.8.4 Subroutine RNMFND

PURPOSE: Find record type number for BOOT

ENTRY POINTS: RNMFND

FORMAL PARAMETERS: RQNAME: Record type name
RQNUMB: Output record type number

COMMON BLOCKS: C10, C20

SUBROUTINES CALLED: NEXTTT

CALLED BY: BOOT

Method:

ISW is set to 1 to show this is the first pass. NEXTTT is called on the RCTYP chain until either the record type name retrieved (RNAME) is equal to RQNAME or the header of the chain is retrieved. If a match is found RQNUMB is set to RNUMB. If the header is retrieved ISW is checked. If ISW=1, it is set to 2 and the search of the RCTYP chain begins anew. If ISW=2, RQNUMB is set to zero.

Subroutine RNMFND is illustrated in figure 25.

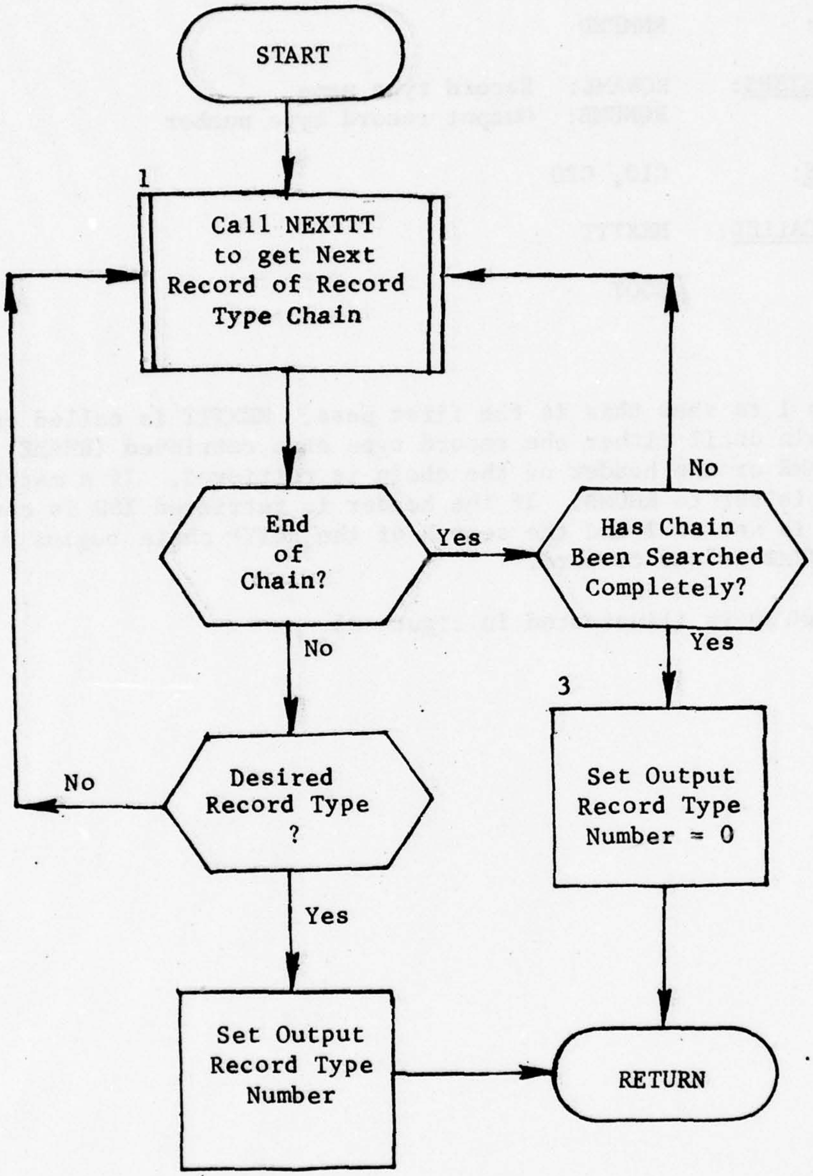



Figure 25. Subroutine RNMFND

3.8.5 Subroutine SEEKER

| | | |
|---|---|---|
| PURPOSE: | Look for particular record on a chain | |
| ENTRY POINTS: | SEEKER | |
| FORMAL PARAMETERS: | RTYPE: | Record type of sought record |
| | CHAIN: | Chain name |
| | VALUE: | Identifying value |
| | FIELD: | Identifying attribute's address |
| | BRANCH: | =1, found; =2, not found |
| COMMON BLOCKS: | C10, C30 | |
| SUBROUTINES CALLED: | NEXTTT | |
| CALLED BY: | BOOT | |

Method:

Set BRANCH to show record not found. NEXTTT is called on chain CHAIN until either a record is retrieved whose type is not RTYPE or a record is retrieved where the value of MAIN (FIELD) is VALUE. In the first case the routine returns. In the second case BRANCH is set to show record was found.

Subroutine SEEKER is illustrated in figure 26.

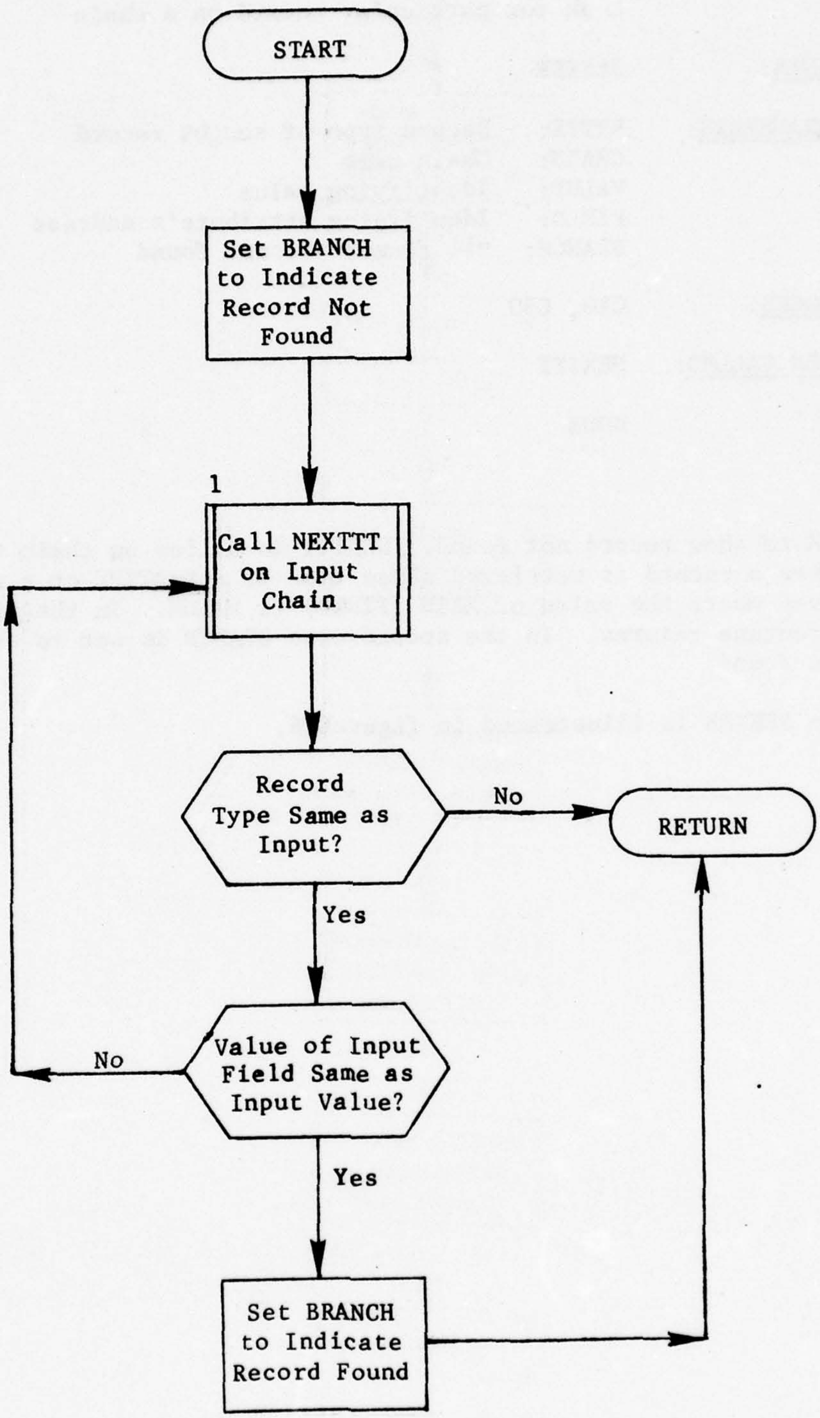



Figure 26. Subroutine SEEKER

3.8.6 Subroutine STRMAK

| | |
|---|---|
| PURPOSE: | Create character string from two fields |
| ENTRY POINTS: | STRMAK |
| FORMAL PARAMETERS | STRING: Output string (12 characters)<br>FIELDA: First field (8 characters)<br>FIELDB: Second field (8 characters) |
| COMMON BLOCKS: | None |
| SUBROUTINES CALLED: | None |
| CALLED BY: | BOOT |

Method:

The first six characters of FIELDA are stored in the first word of STRING. The last two characters of FIELDA are stored in the first two characters of the second word of STRING. Finally, the last four characters of the second word of STRING are set to the first four characters of FIELDB.

Subroutine STRMAK is illustrated in figure 27.

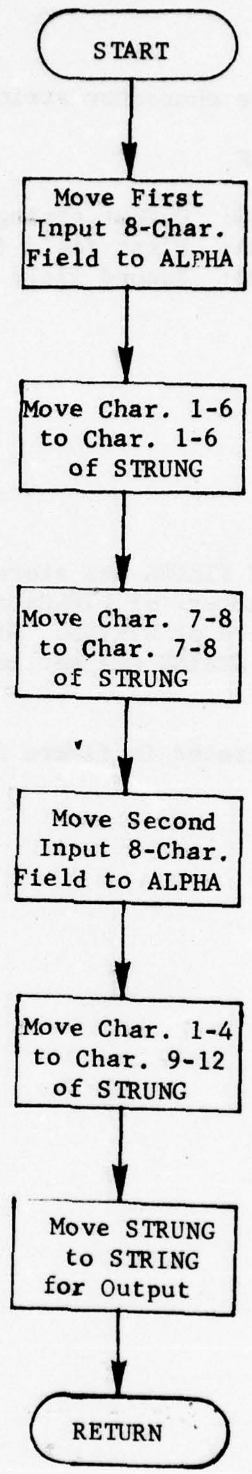



Figure 27. Subroutine STRMAK

### 3.9 Subroutine ERRFND*

PURPOSE: Control syntax analysis process

ENTRY POINTS: ERRFND

FORMAL PARAMETERS: None

COMMON BLOCKS: FIRST, IPQT, STRING, SYMBOL, TABLZ

SUBROUTINES CALLED: GETSTR, LNGSTR, SYNTAX, TABINS, WEBSTR

CALLED BY: COP

Method:

First the ERRFND table counts are set to zero and the SPCIAL switch is set to 'False' to indicate that the previous symbol is not an operator. For every input string but the first, GETSTR and WEBSTR are called. (INICOP has already called them in the case of the first sentence and for every other sentence the verb has been read.) If the call to GETSTR encounters an end of input (ENDSW), WEBSTR is not called. If the input string is a null (Type=11), GETSTR is called again.

Whether the first string of the sentence or not, process arrives at statement 3 (see figure 28). If the type is a long string (Type=2) LNGSTR is called. Next SYNTAX is called to check for syntax errors. If end of input, process stops here. Otherwise, the symbol is begun (KYMBOL) by setting it to TYPE. Then the rest of the symbol is created according to its type. Operators, adverbs, special words and verbs (which are at the beginning of the sentence) have their identifier values added to ther symbol and TABINS is called to save the symbol. Long strings have already had their symbols created. Attributes, alphabetics and numerics have their values stored by one call to TABINS and their symbols stored by another.

After a string's symbol is stored, the process returns to call GETSTR for the next string until an end of input or a second verb.

Subroutine ERRFND is illustrated in figure 28.

---

*Main routine of overlay ERRF

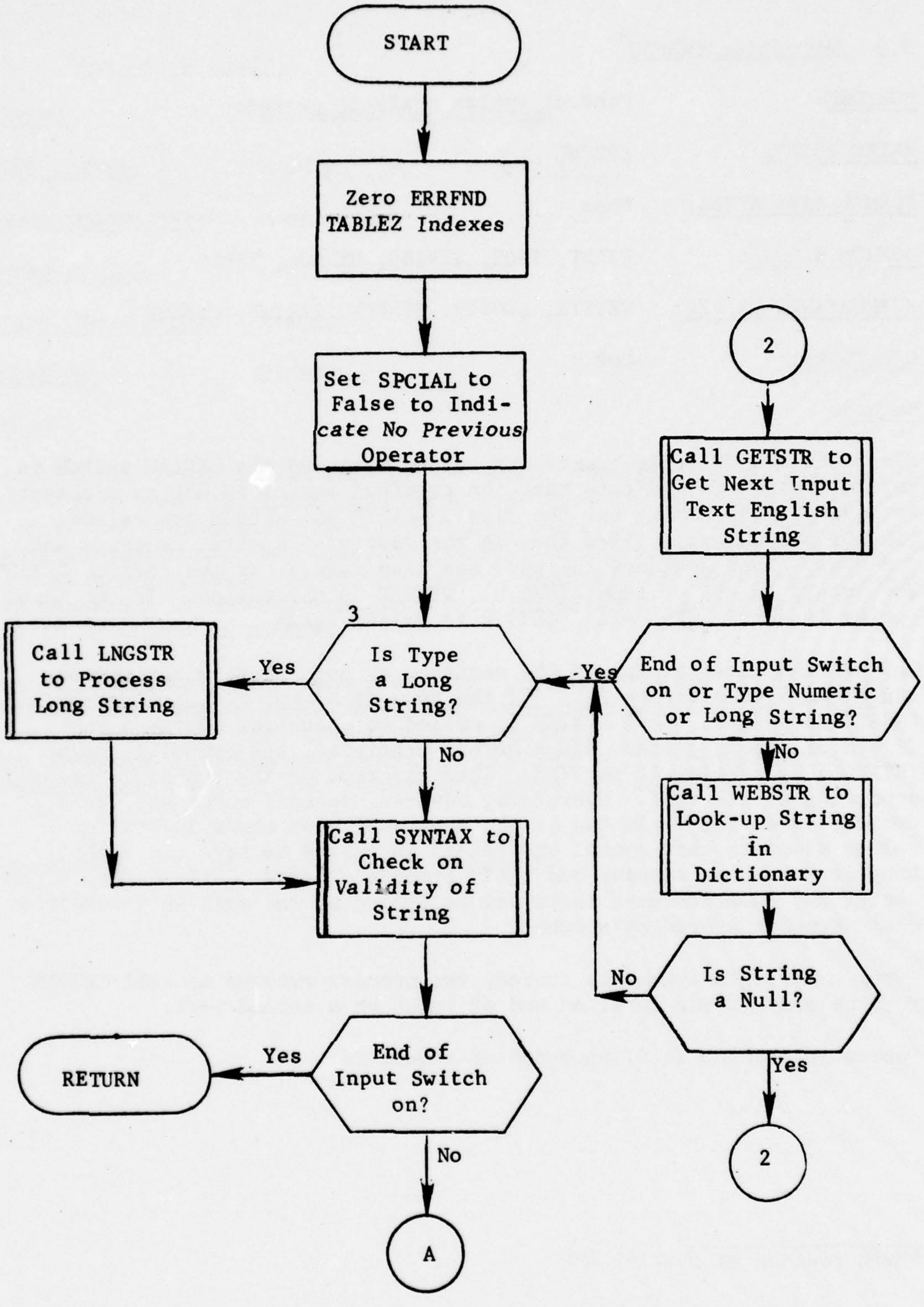



Figure 28. Subroutine ERRFND (Part 1 of 2)

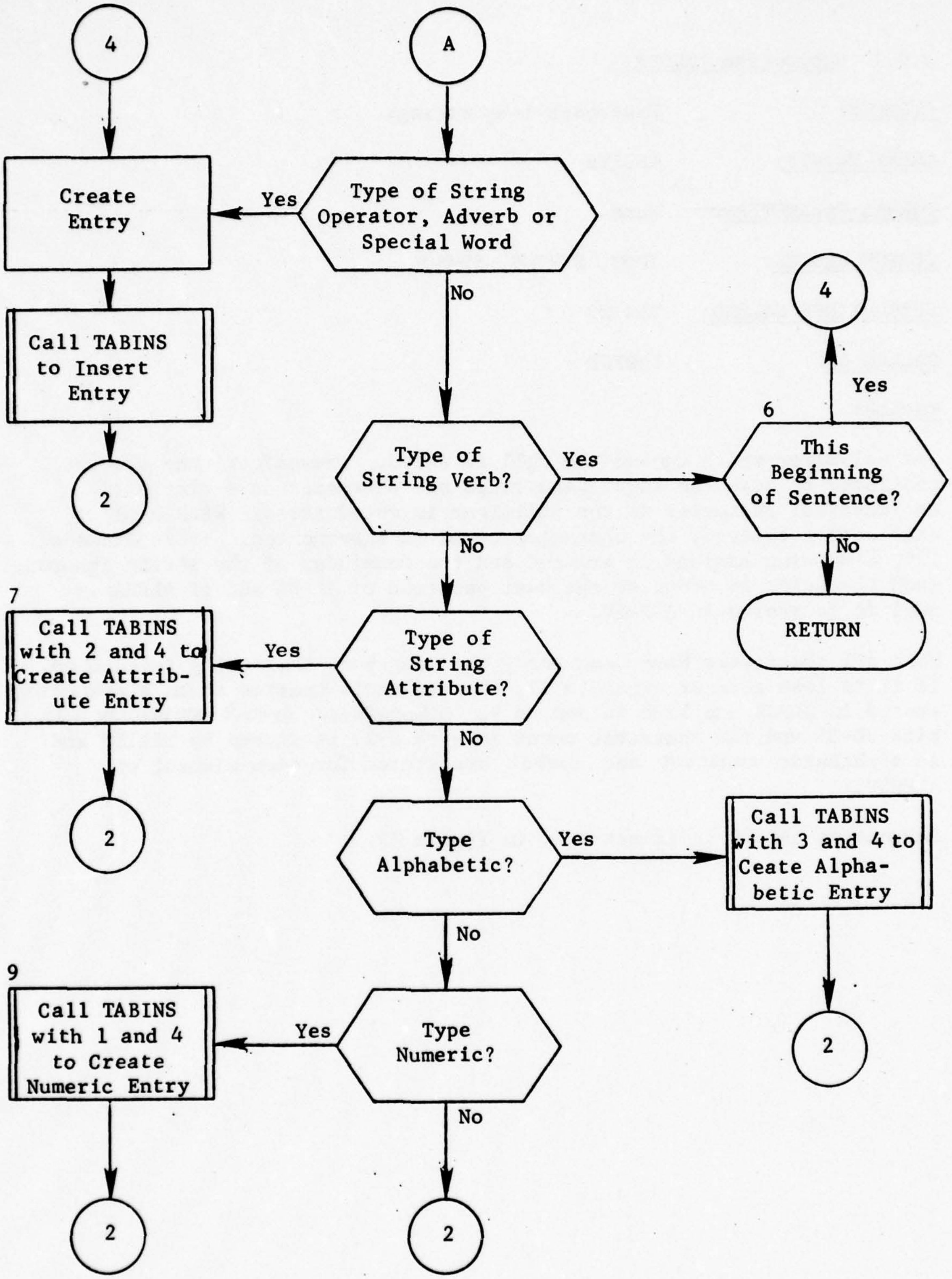



Figure 28. (Part 2 of 2)

### 3.9.1 Subroutine LNGSTR

| | |
|---|---|
| PURPOSE: | Processes long strings |
| ENTRY POINTS: | LNGSTR |
| FORMAL PARAMETERS: | None |
| COMMON BLOCKS: | IPQT, STRING, SYMBOL |
| SUBROUTINES CALLED: | TABINS |
| CALLED BY: | ERRFND |

Method:

The delimiter which caused the call is saved. Thereafter, the process continues to scan the input card image one character at a time until an identical character to the delimiter is encountered. With each character retrieved, the character count is incremented. If it exceeds 120, a warning message is printed and the remainder of the string ignored. Each character is added to the next position of ALPHA and if ALPHA is full it is stored in ALPHSV.

When all characters have been read, the length of the string is checked. If it is less than or equal to 12, the string is treated as an alphabetic, stored in ALPHA and TYPE is set to 9. Otherwise, a symbol containing 2 in bits 30-35 and the character count in bits 0-29 is stored by TABINS and an alphabetic constant and symbol are stored for each element of ALPHSV.

Subroutine LNGSTR is illustrated in figure 29.

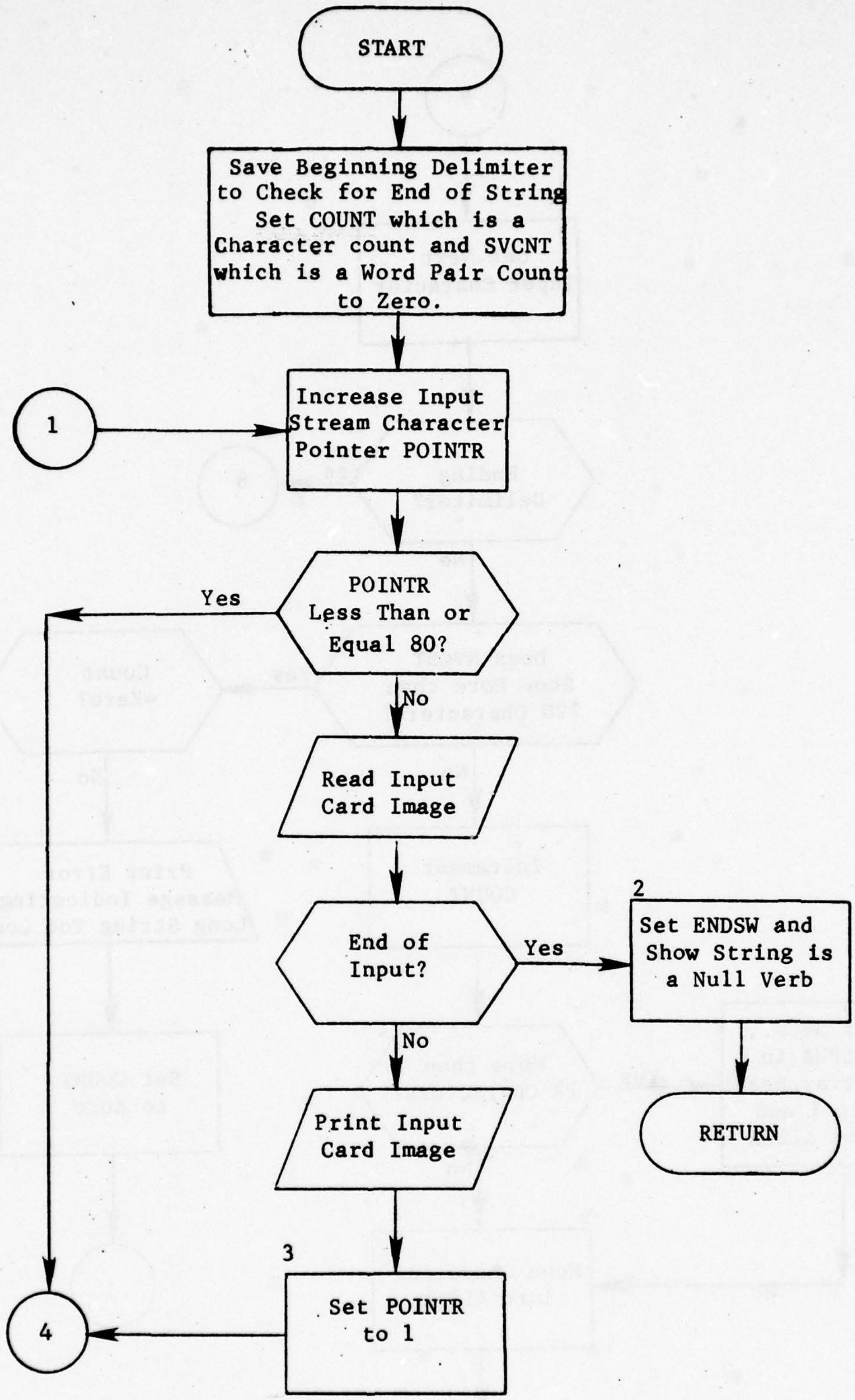



Figure 29. Subroutine LNGSTR (Part 1 of 4)

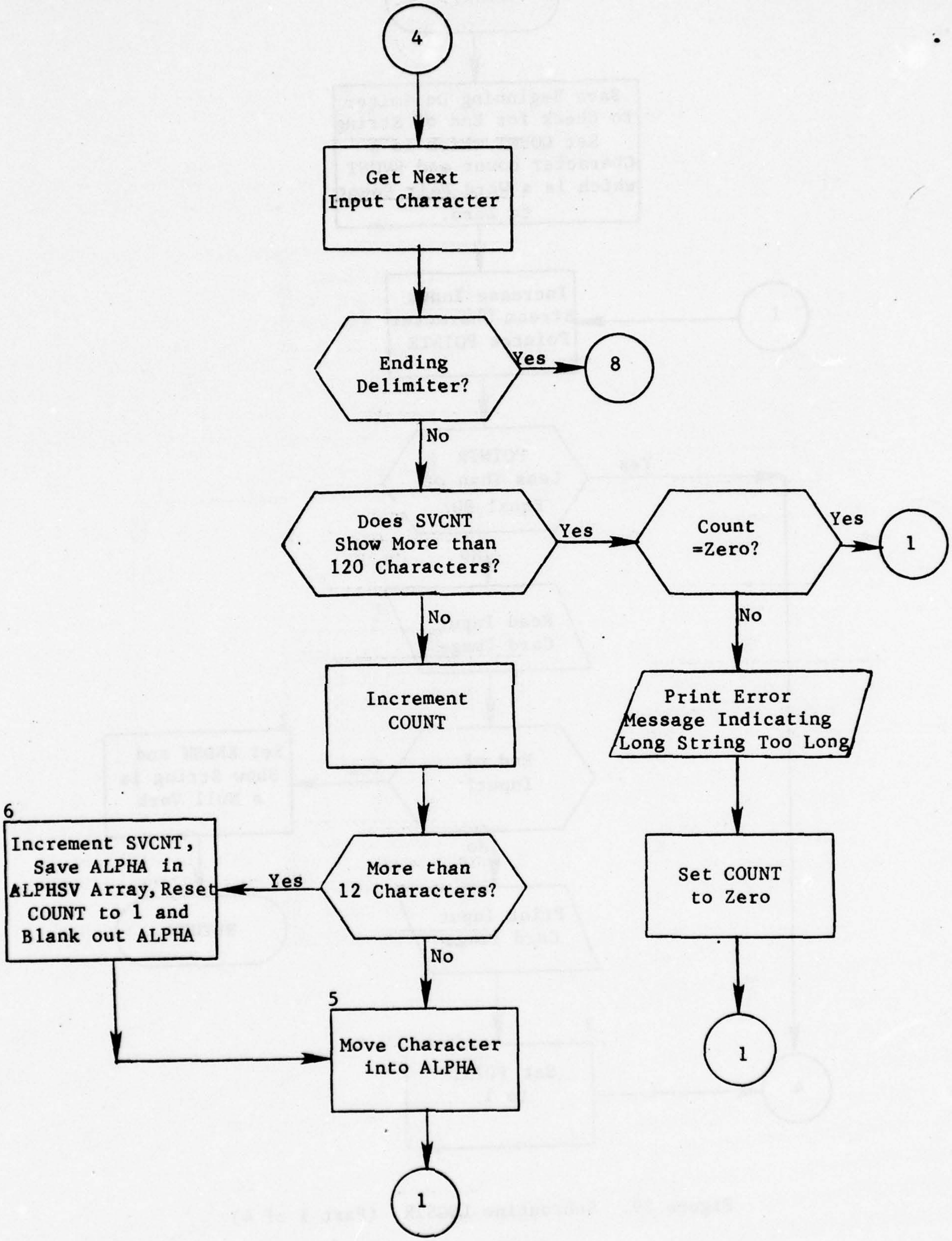



Figure 29. (Part 2 of 4)

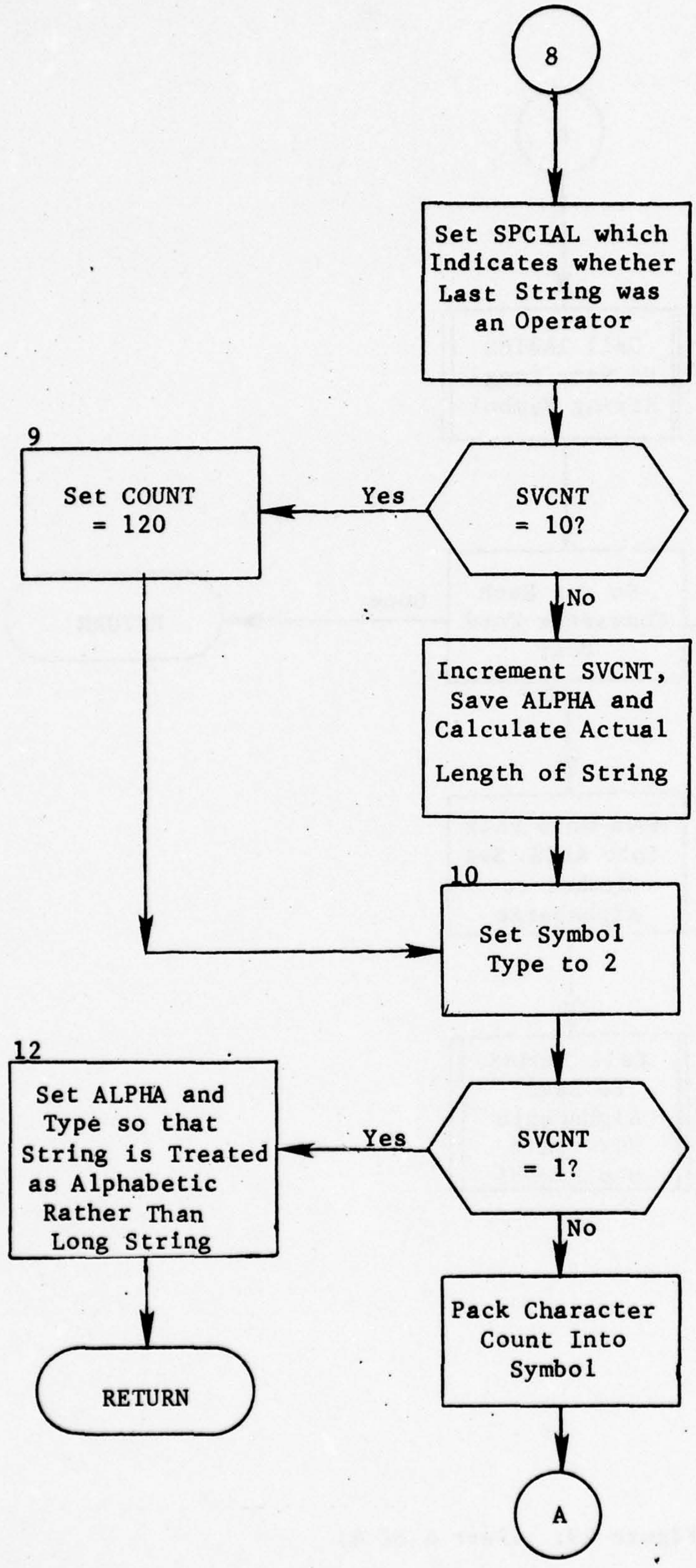



Figure 29. (Part 3 of 4)

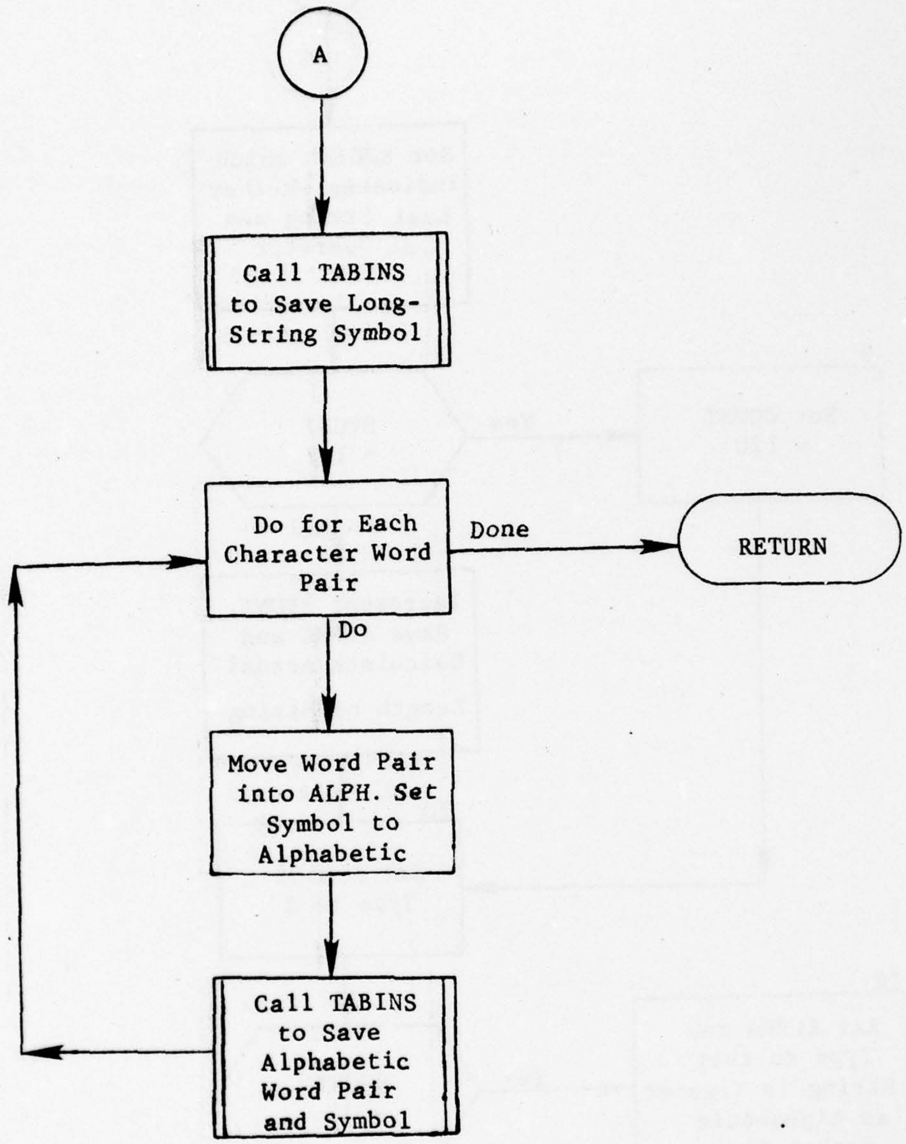



Figure 29. (Part 4 of 4)

## SECTION 5. EDITDB MODULE

### 5.1 General Purpose

The EDITDB module performs two similar but quite distinct functions. First, it performs the standard consistency checks as outlined in table 13 in Users Manual, Volume I. Second, it will perform any set of non-standard edits desired by the user. This second function allows the user to check the validity of data attributes not covered by the standard edits or make certain that attributes fall within any desired limits other than those listed in the directory.

### 5.2 Input

The input to EDIT consists of the cards containing the generalized text english commands as described in the Users Manual. The data base can be in any stage completion after it has been initialized.

### 5.3 Output

EDIT will create no tapes or new data base records.

### 5.4 Concept of Operation

The execution of EDIT always performs consistency checks and determines counts of various subsections of the data base to ensure that limitations as dictated by the QUICK system are not exceeded. In addition, EDIT performs nonstandard checks as introduced by the user through the WHERE, FIELDS or WITH clauses. These generalized clauses allow for special subsections of the data base to be checked against ranges for validity as directed.

EDIT consists of four overlays each of which conducts a special function. The main objectives are met within these overlays.

### 5.5 Identification of Subroutine Functions

5.5.1 Subroutine COUNTS. This subroutine (or overlay) queries major portions of the defined data base and produces summary counts of various collections contained within the data. The processors to be executed within the QUICK system, uses local dimensioned arrays for efficient processing and, hence, certain limitations must be met. This subroutine, then, informs the user if certain subsections (geography, weapon types, and groups, etc.) exceeds any of these constraints. The code counts entries in a straightforward fashion.

VALCT - Value element branch
  1 - expecting end of element or OF
  2 - expecting identifying attribute
  3 - expecting alphabetic or numeric constant
VALEL - True if processing value element
VALQE - True if processing value expression
VALTYP - True when expecting new value element or left parenthesis
VLPRCT - Value expression parenthesis count

Processing proceeds as follows:

First the string is checked to see if it is a verb, if not, branch to statement 5 (see figure 30). If so INCOM is checked. If INCOM is true but ENDCOM is false an error has occurred. Otherwise INCOM is set to false and the subroutine exists. If INCOM is false, the VERB chain is searched to find a match. When the match is found INCOM and ENDCOM are set to two and INCLZ is set to false. If the clause switch indicates that the verb must have a clause (ICSW=1), ENDCOM is set to false.

Statement 5 (figure 30)

If the string is not an adverb, a branch is made to statement 19. If it is an adverb INCOM is checked and if false an error has occurred. Next, if INCLZ is true but ENDCLZ is false an error has occurred. Next, the CLAUSE chain is searched to see if this adverb is matched to the current verb. If it is HEAD is called to relieve the adverb's record and the clause and phrase type (IXTYP and ILTYP) are packed into the adverb's value. Now various switches are set depending on the clause type (IXTYP). If boolean, set SINGE to false, BOOL to true, BLPRCT to zero and BOLTYP to true. If sequence, set SINGLE and BOOL to false. If single set SINGE to true, BOOL to false, SNGCT to zero. If null set INCLZ to false and ENDCLZ and ENDCOM to true. For all but the last, set INCLZ to true and ENDCLZ, ENDCOM and VALQE to false and branch on phrase type (ILTYP). For relational phrases set RESTRC to false, for restricted phrases set RESTRC to true, for both set RELPHR, ELEMNT, EQUALS, LIKE, BETWEN and CONTEQ to false, and PHPRCT to zero. For elemental phrases set ELEMNT to true.

Statement 19 (figure 30)

At this point refer to the flow chart, figure 30 as processing is best illustrated therein. However, some sections that are of particular note follow.

Statement 52 (figure 30)

Here many relational phrases have ended. The LIKE phrase has one more element--the value for the identifiers attribute. The BETWEEN phrase may have an optional 'AND'. After this or without it the BETWEN switch

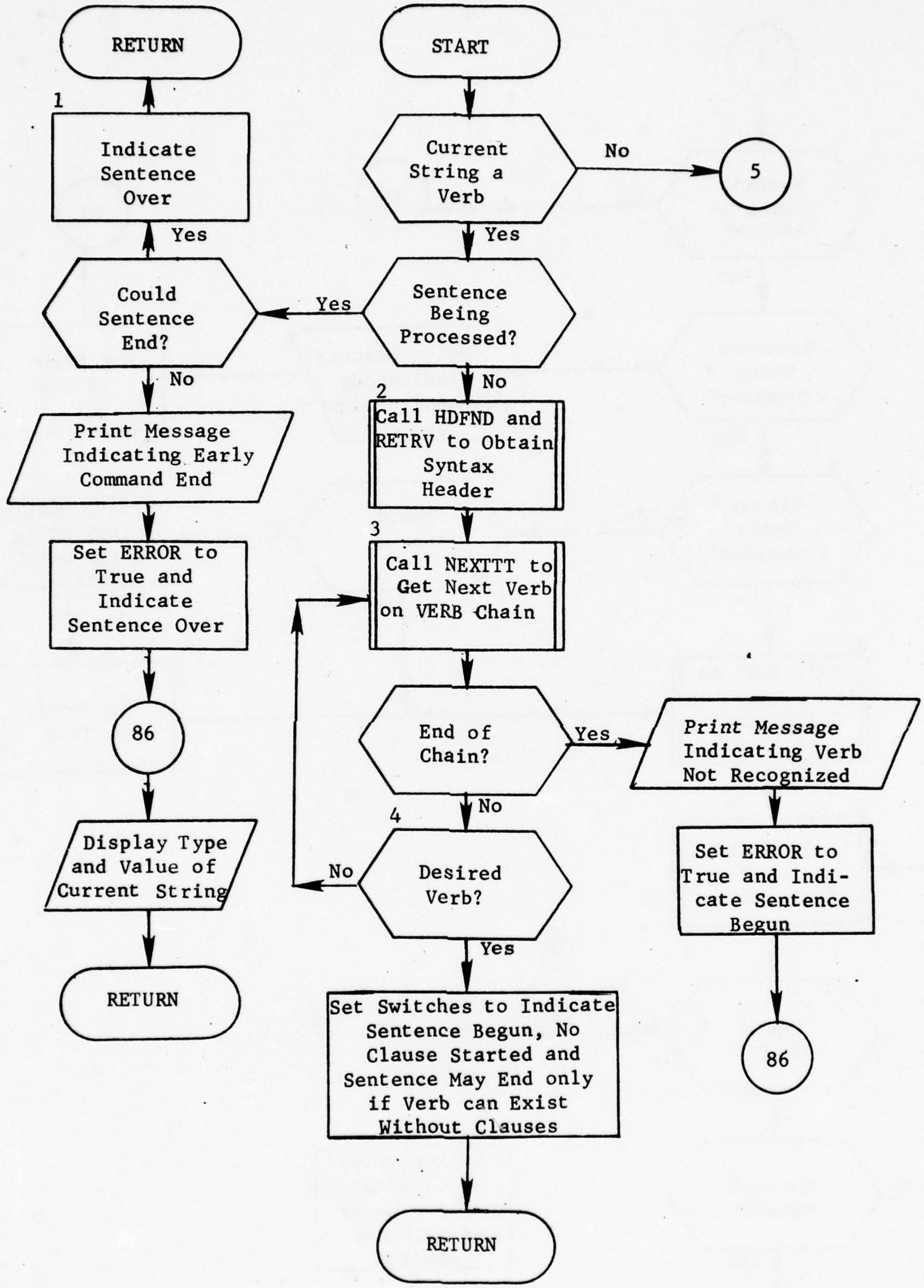



Figure 30. Subroutine SYNTAX (Part 1 of 22)

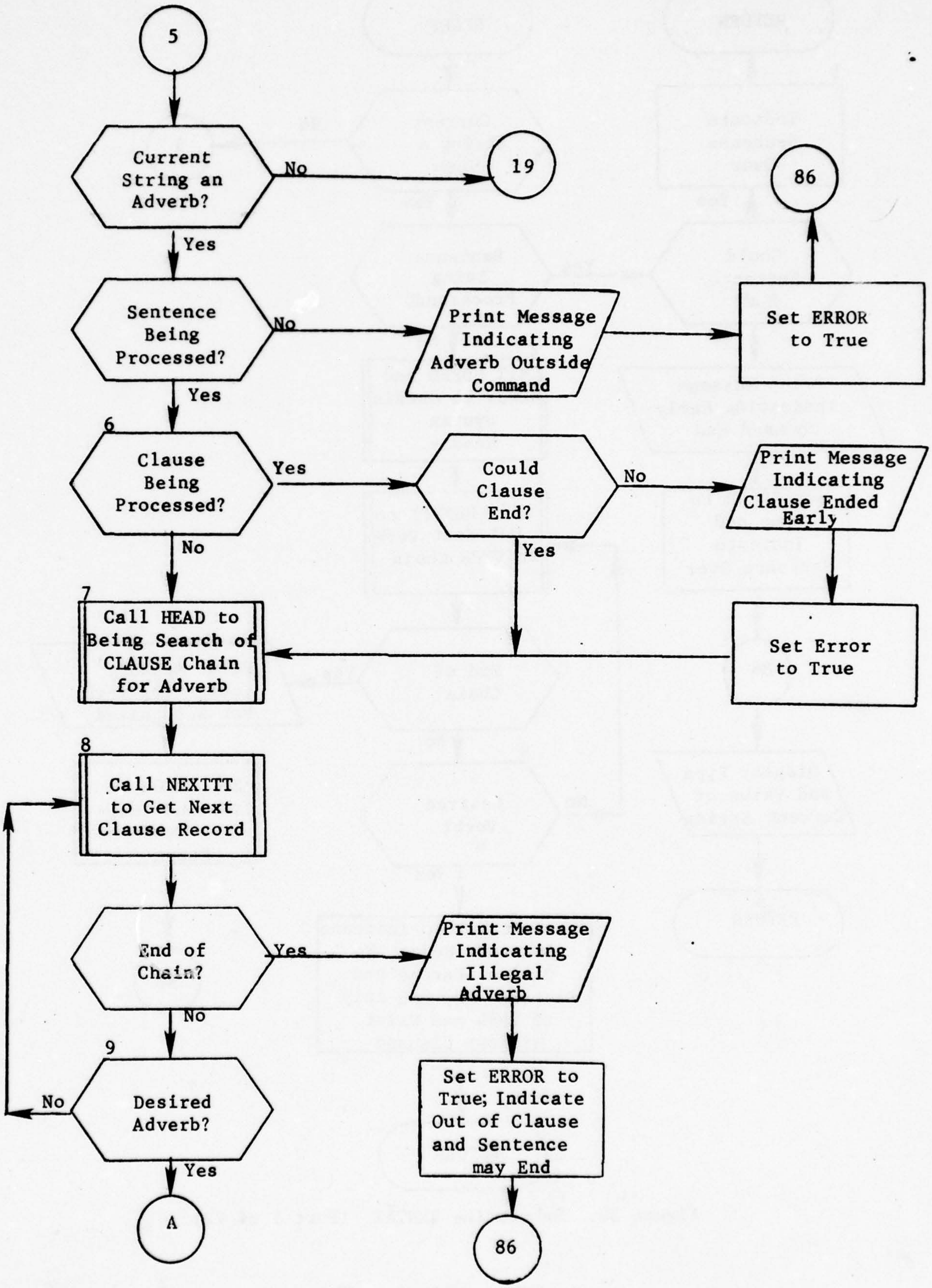



Figure 30. (Part 2 of 22)

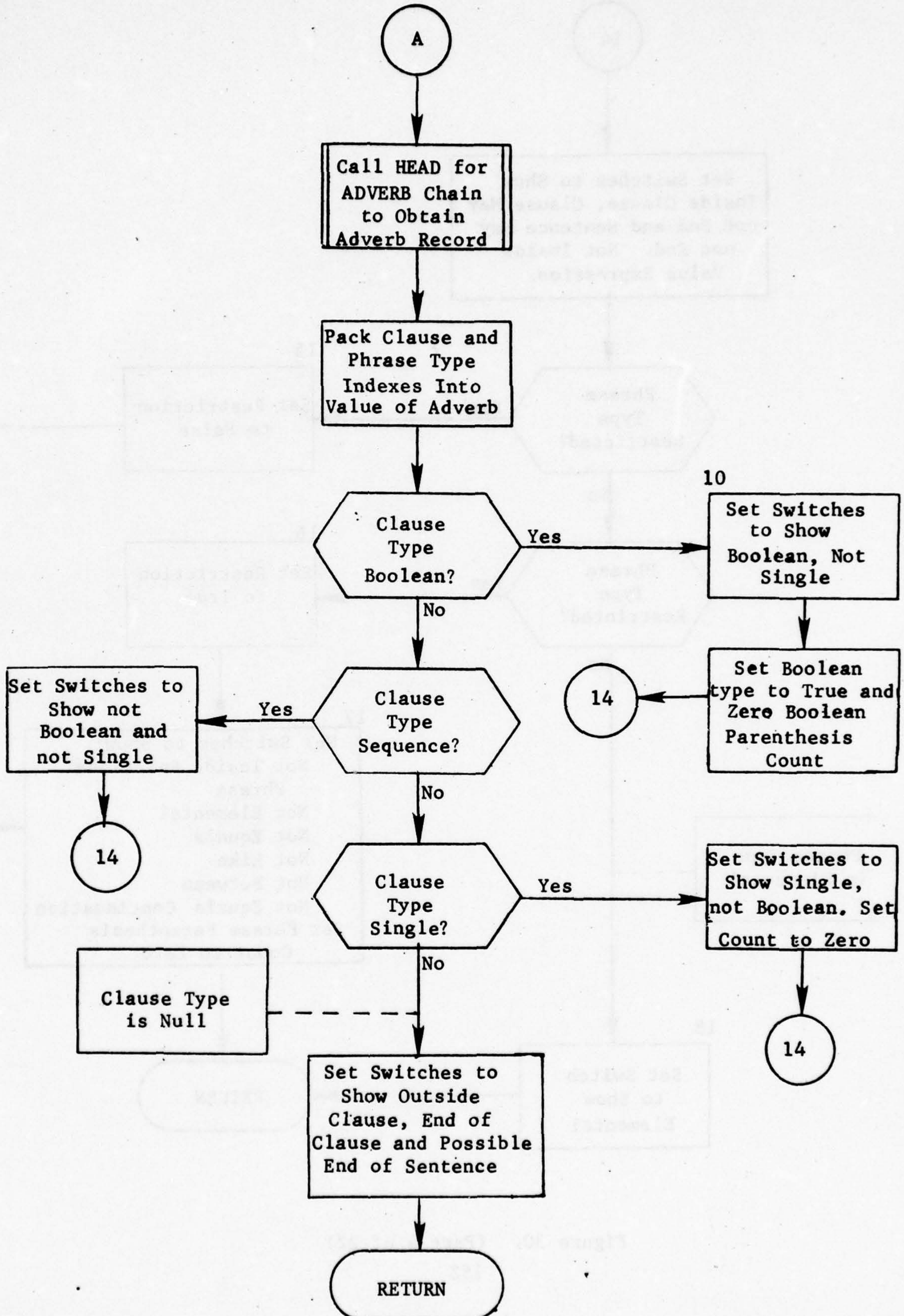



Figure 30. (Part 3 of 22)

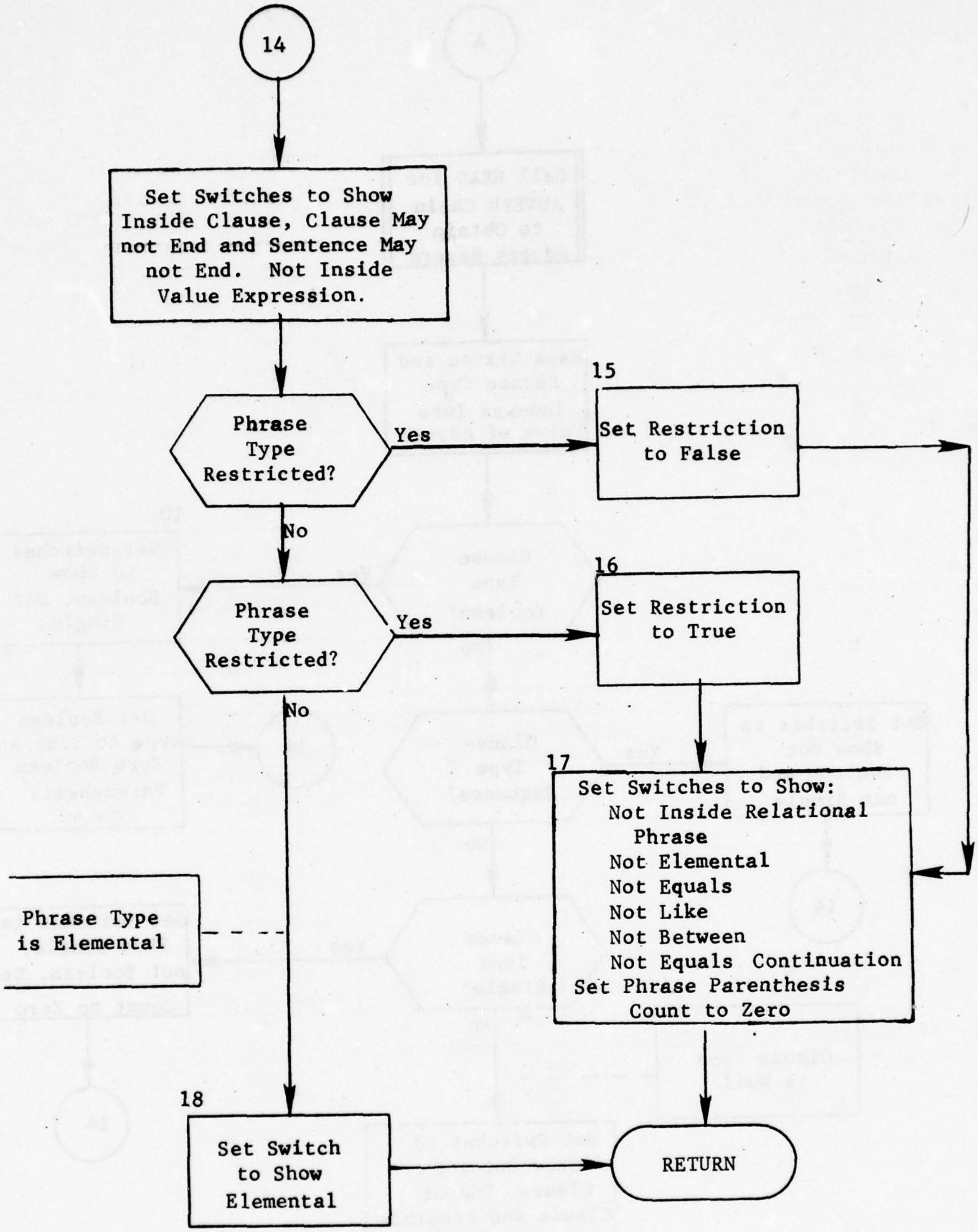



Figure 30. (Part 4 of 22)

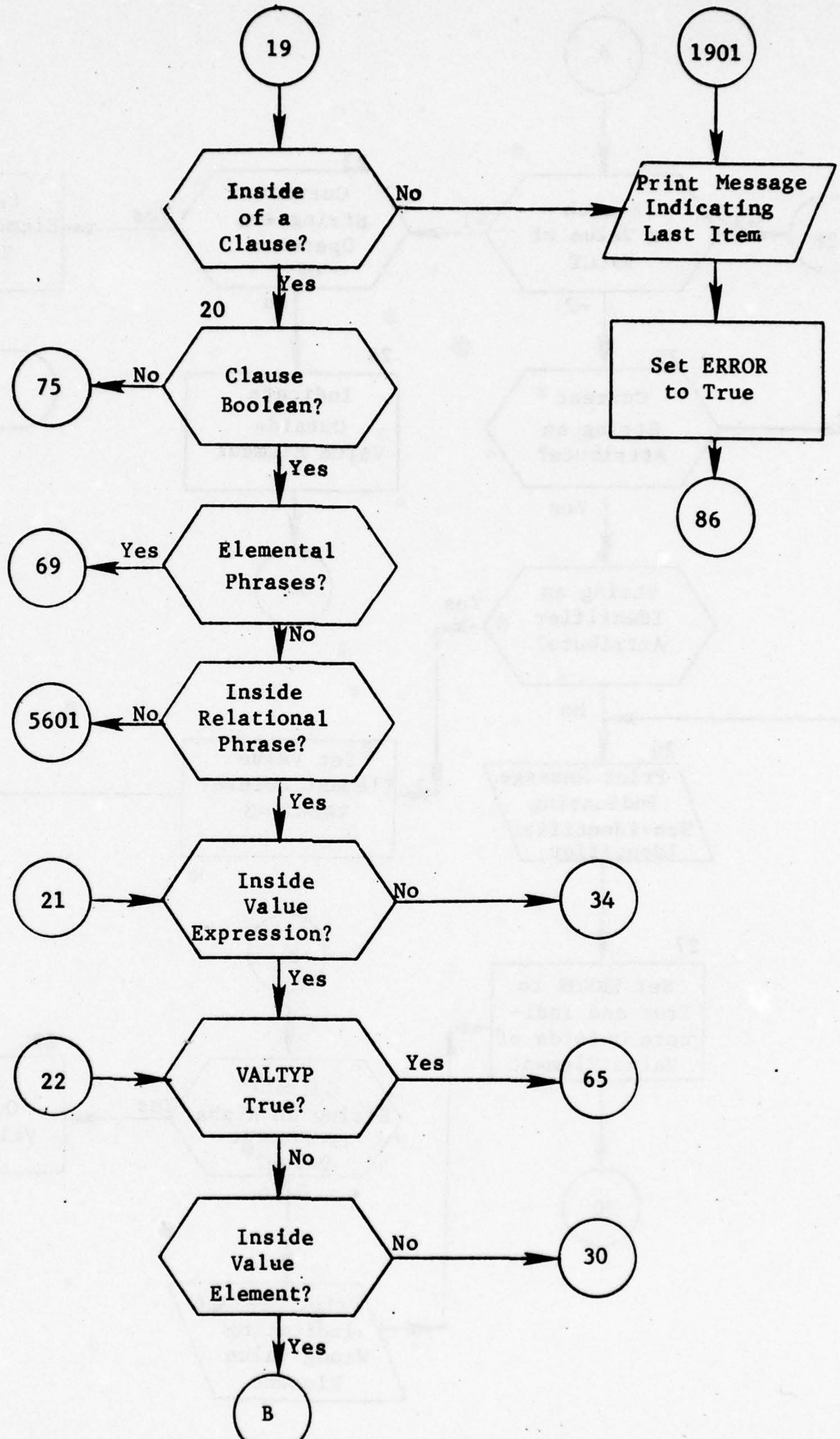



Figure 30. (Part 5 of 22)

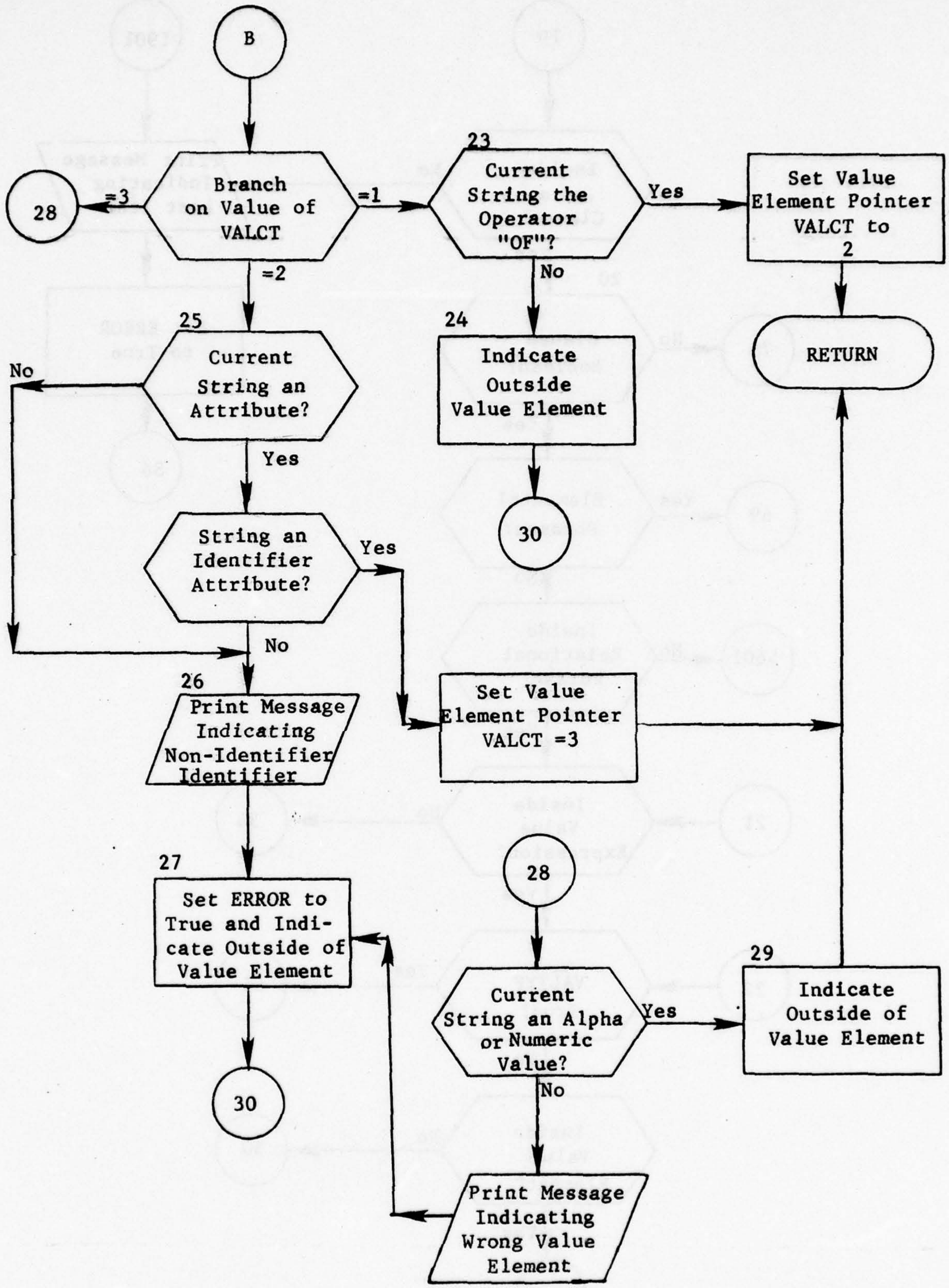



Figure 30. (Part 6 of 22)

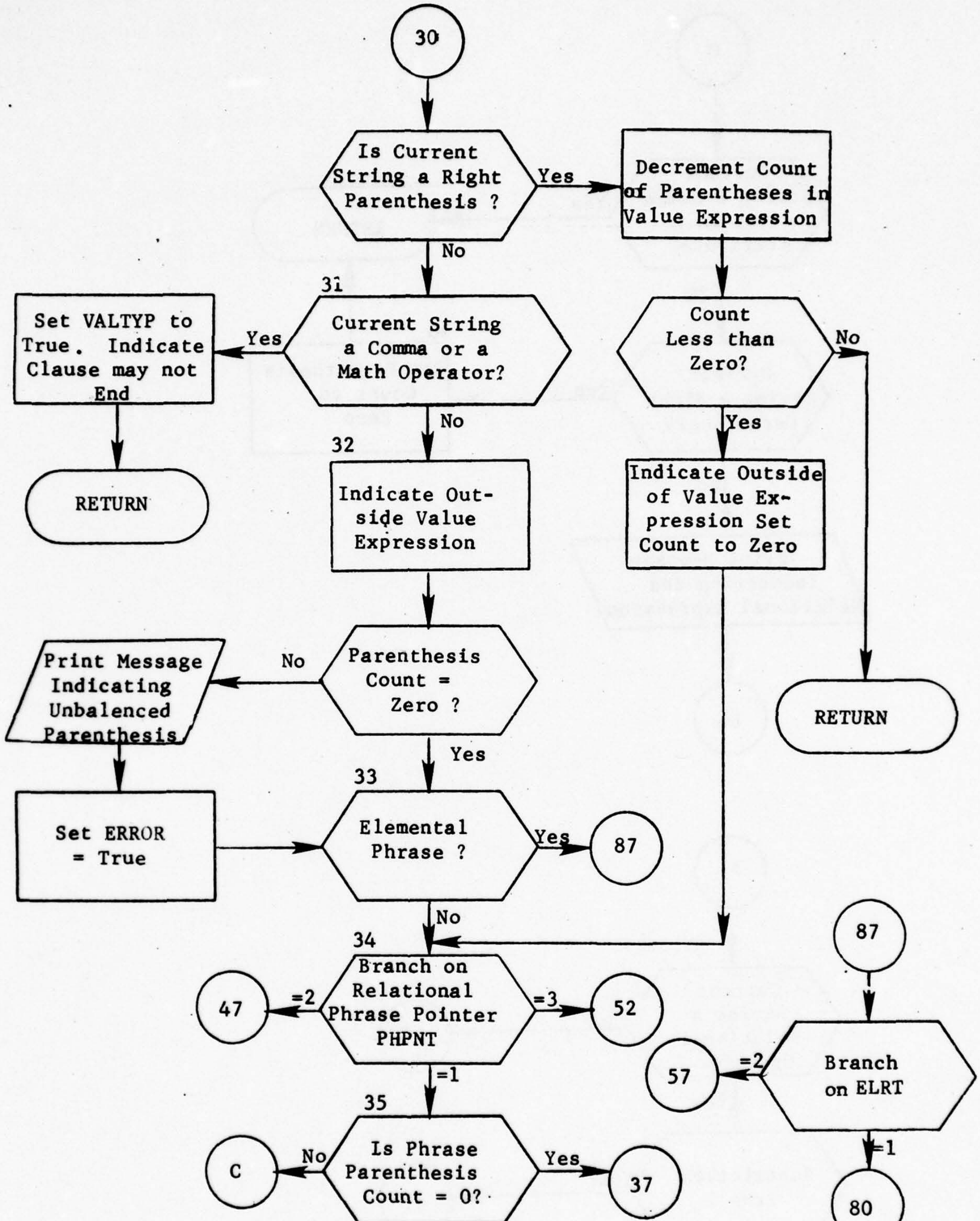



Figure 30. (Part 7 of 22)

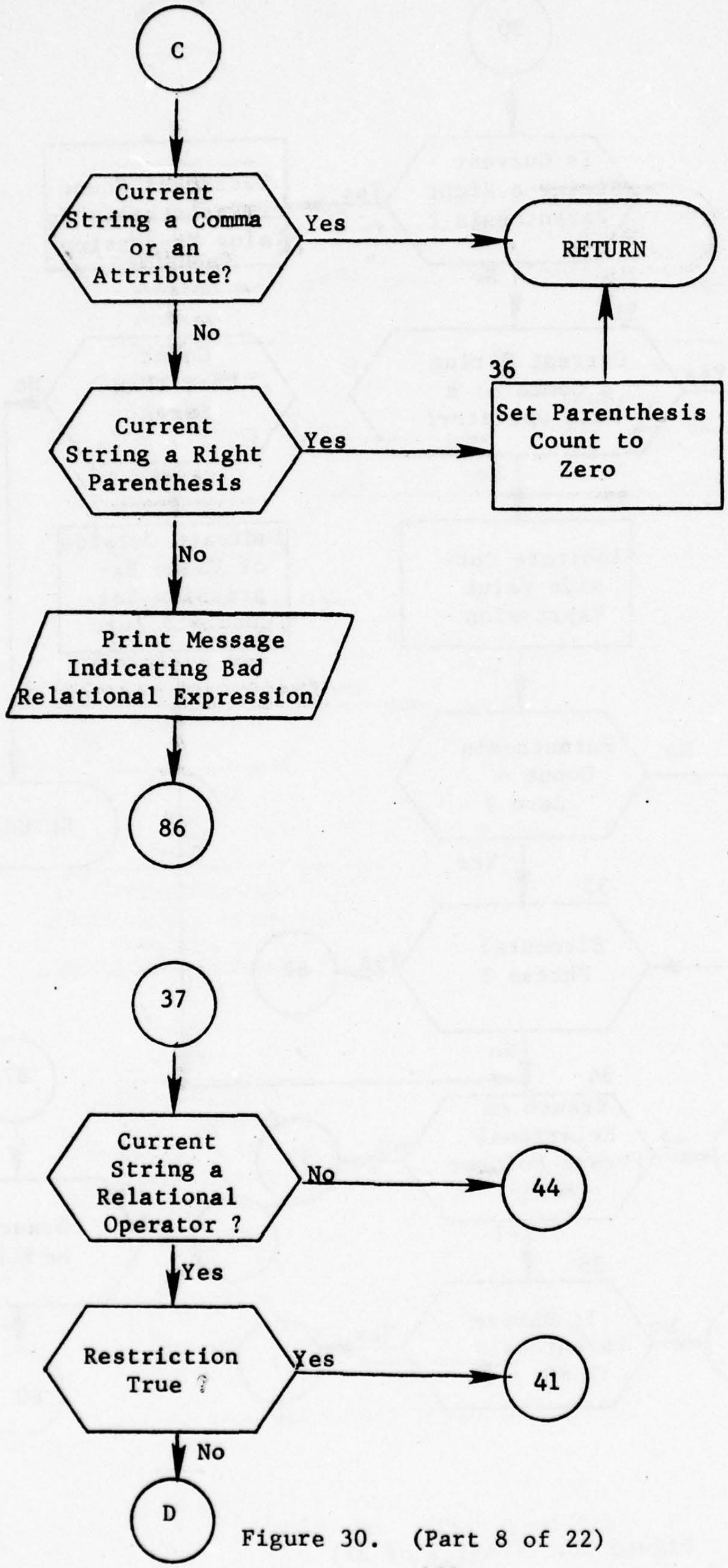



Figure 30. (Part 8 of 22)

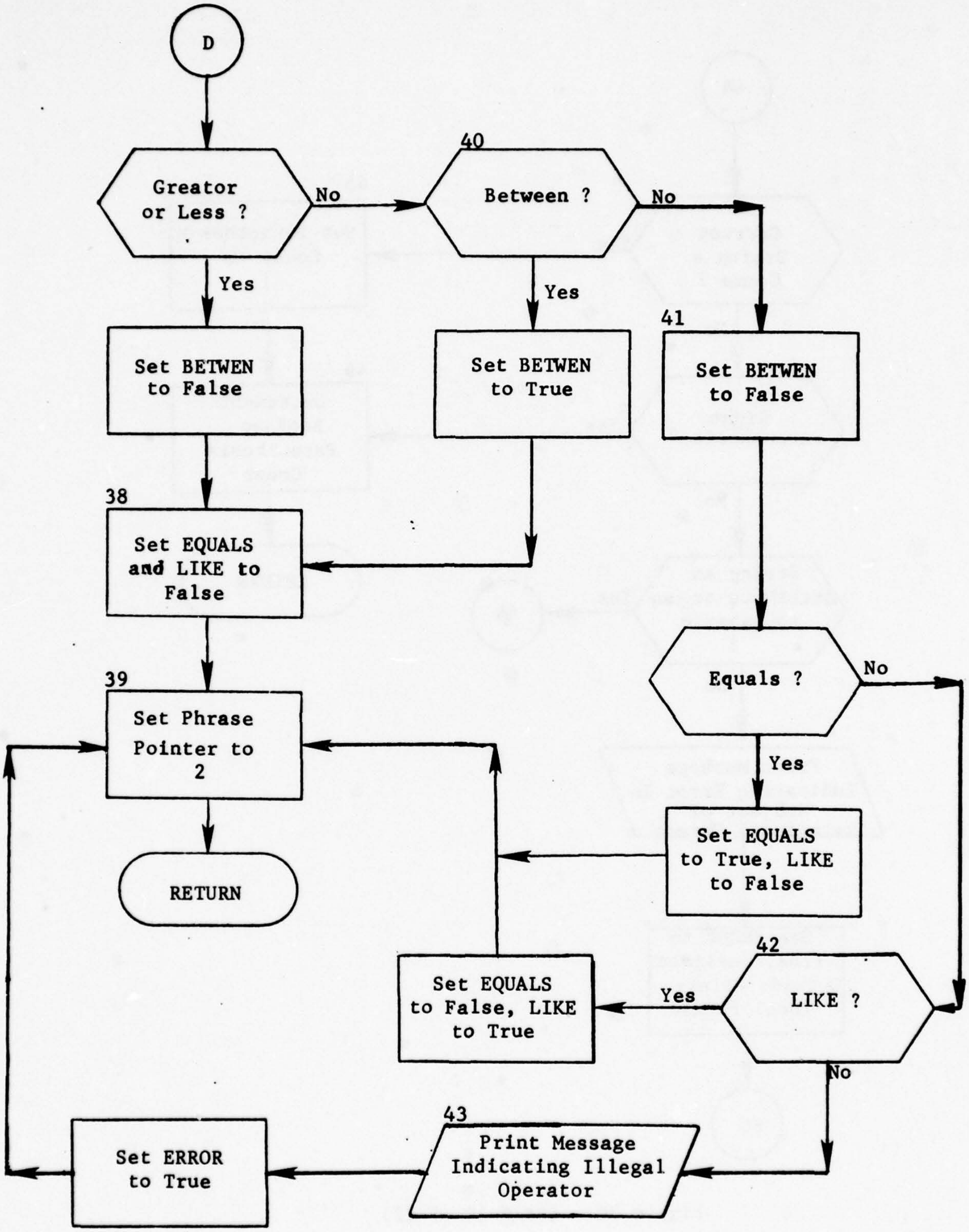



Figure 30. (Part 9 of 22)

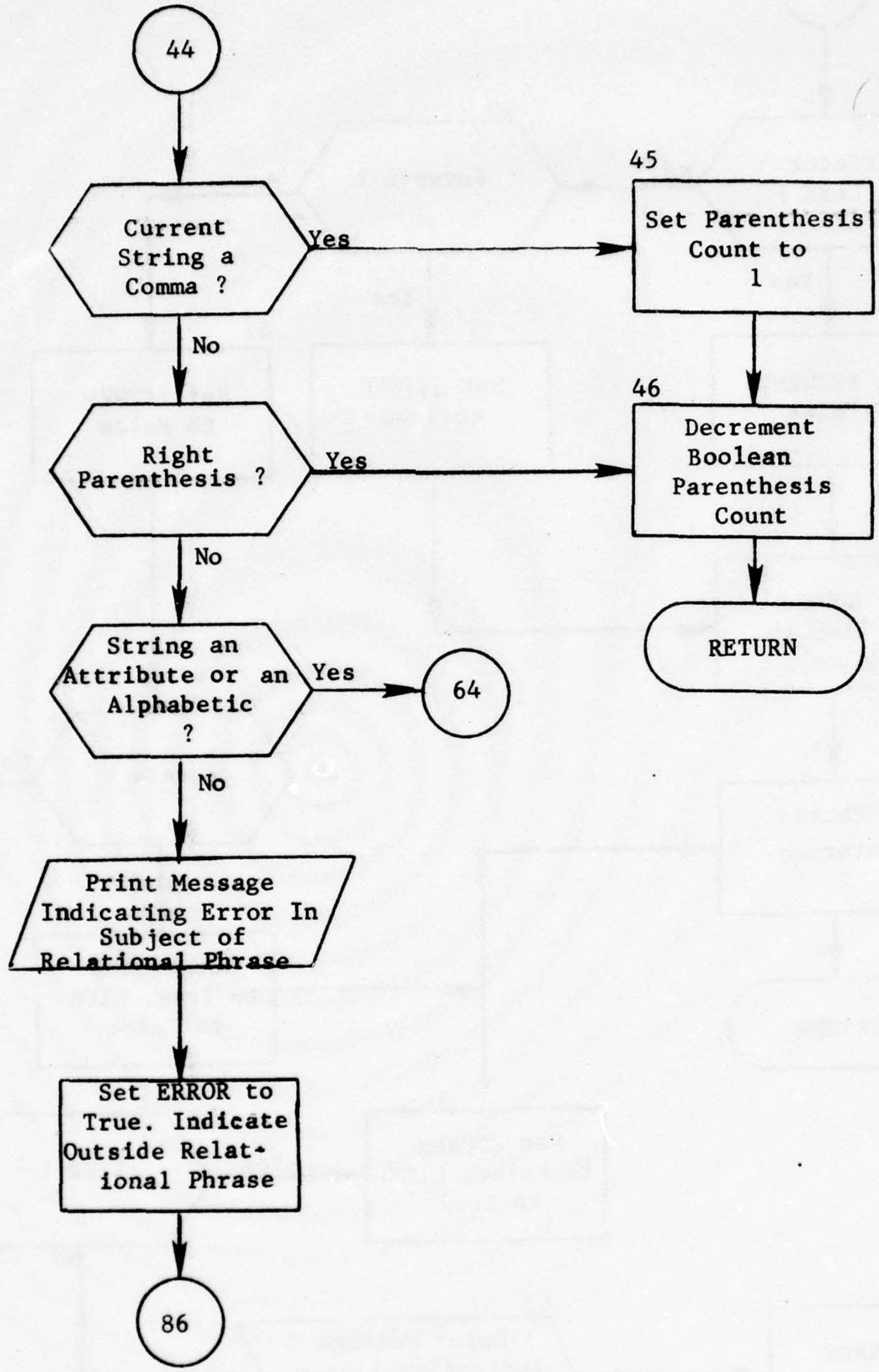



Figure 30. (Part 10 of 22)

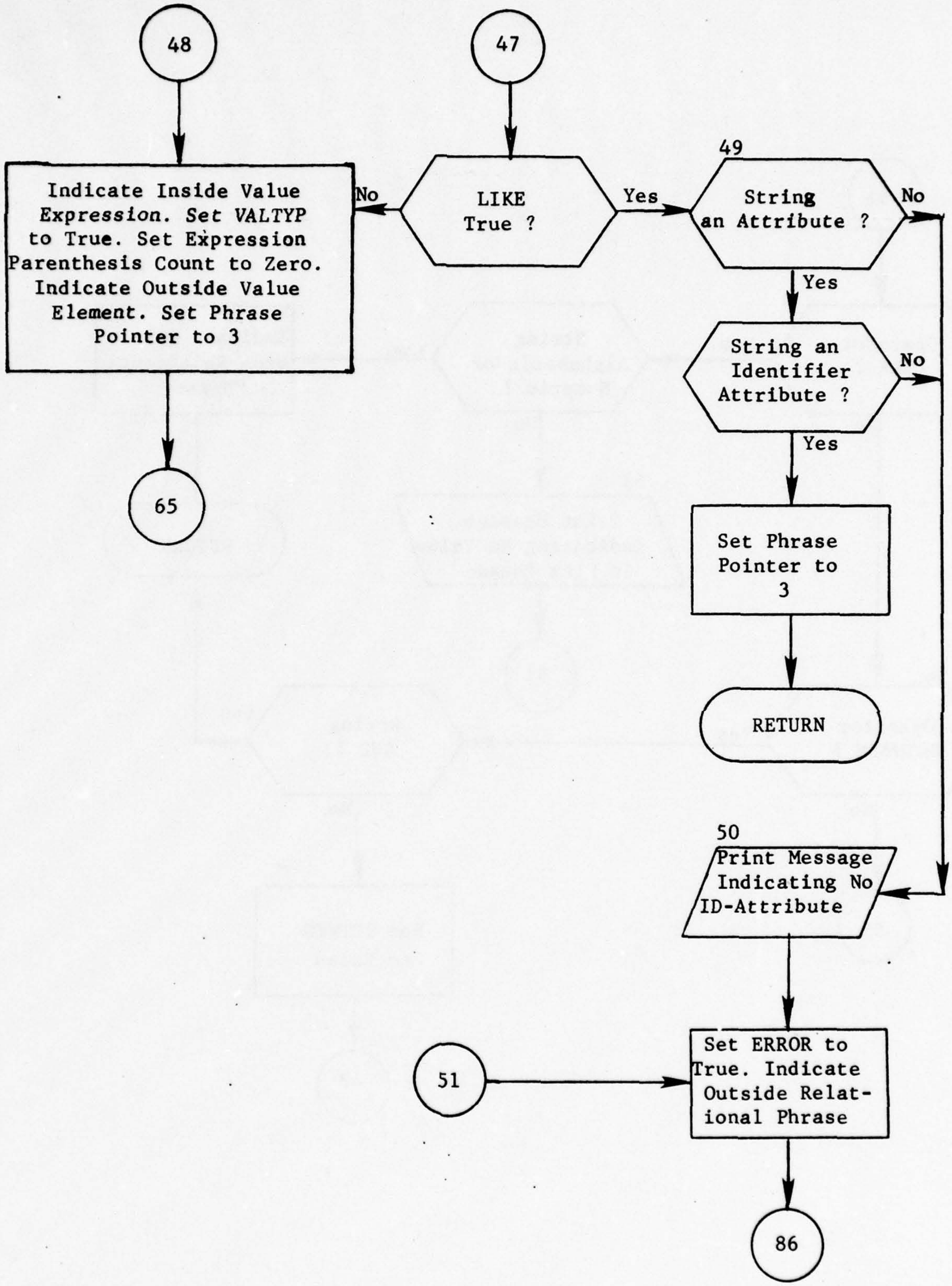



Figure 30. (Part 11 of 22)

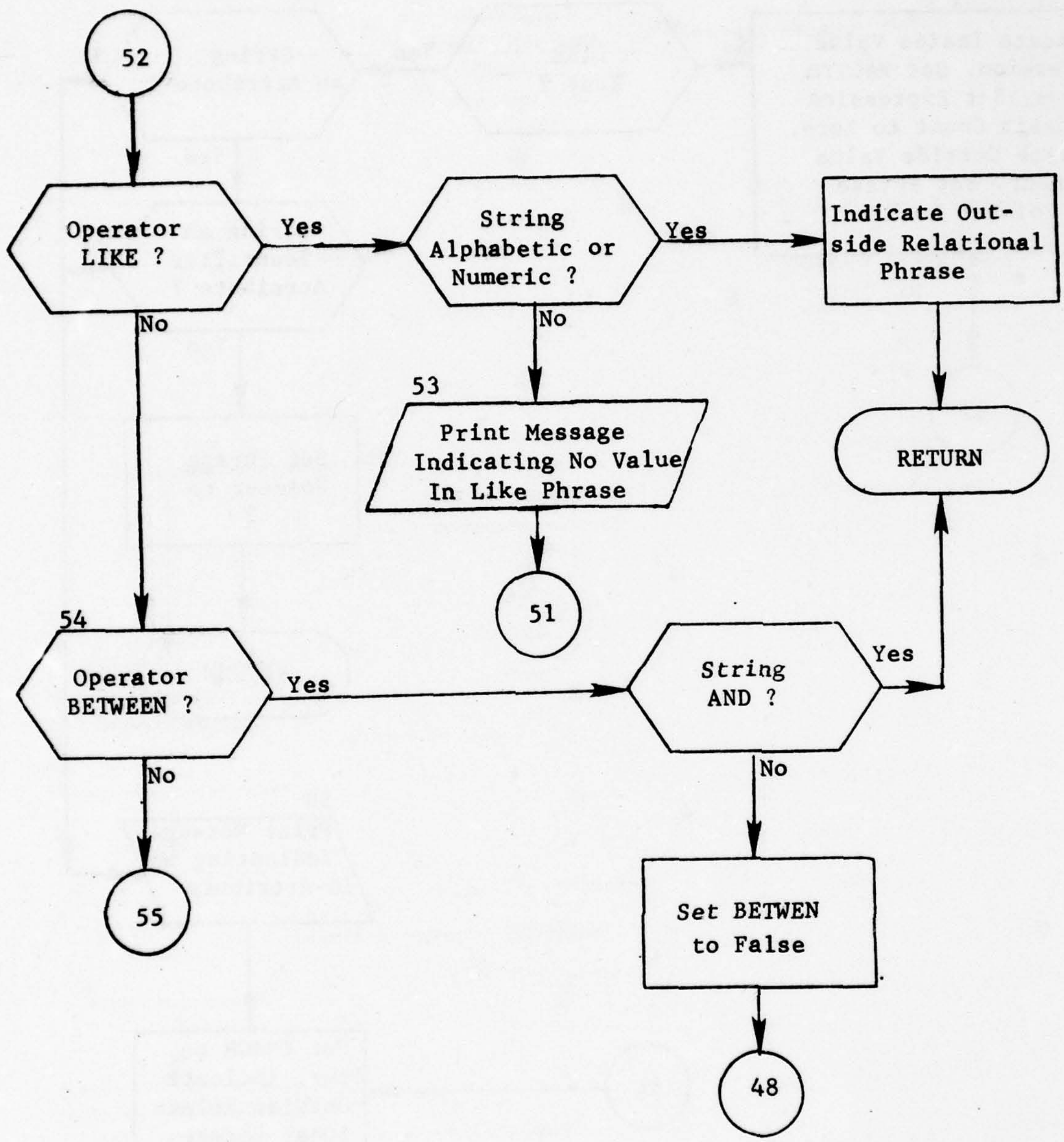



Figure 30. (Part 12 of 22)

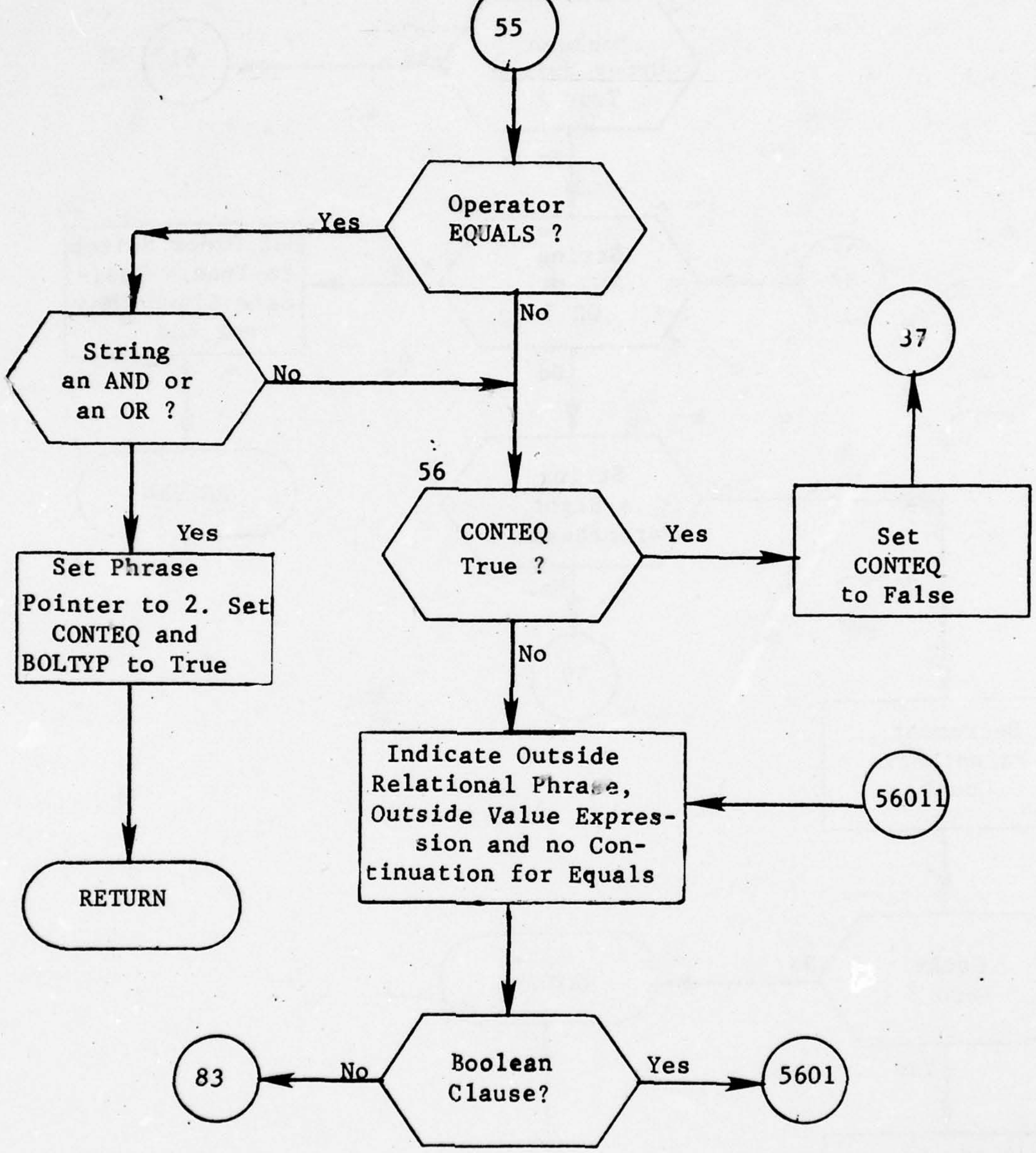



Figure 30. (Part 13 of 22)

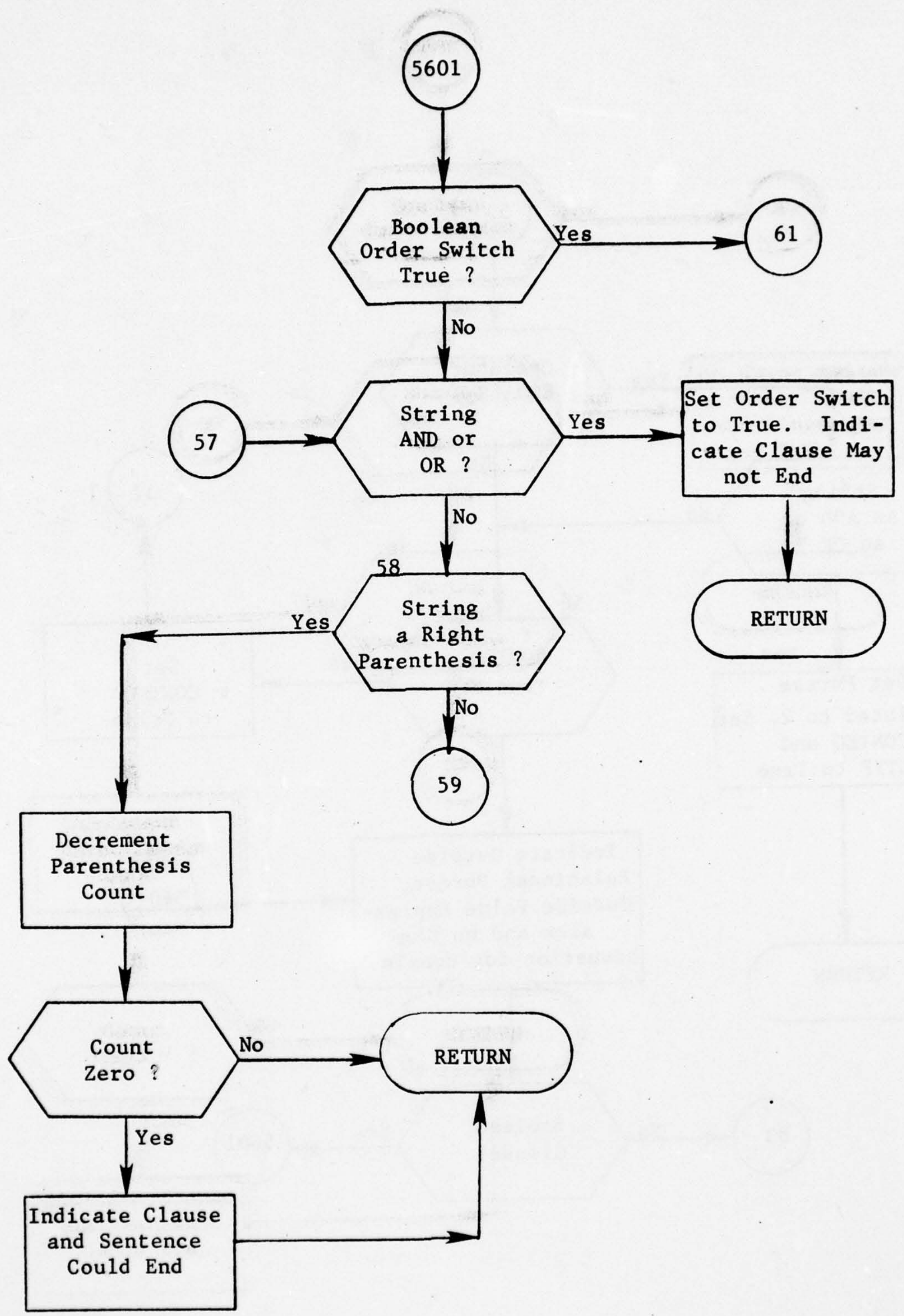



Figure 30. (Part 14 of 22)

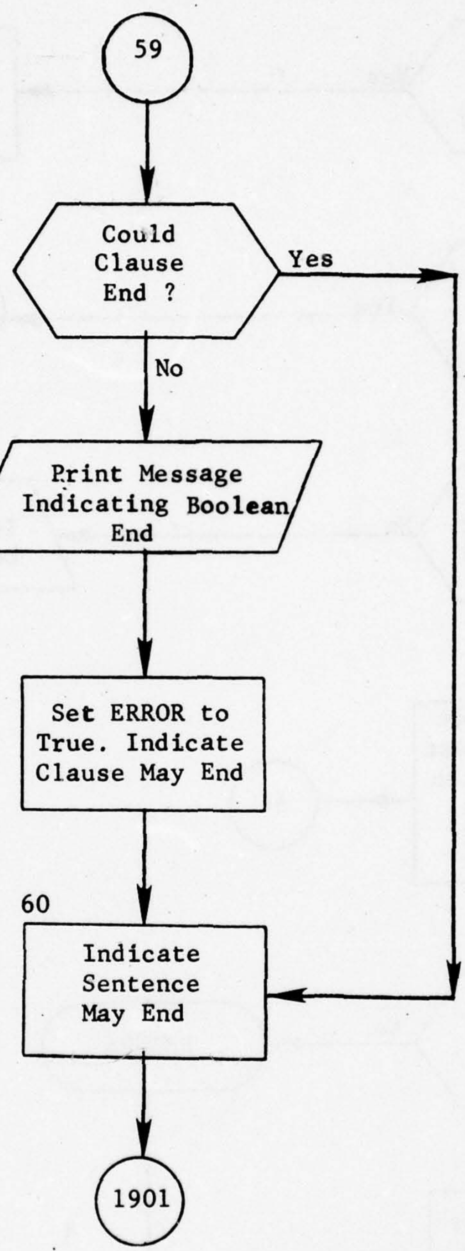



Figure 30. (Part 15 of 22)

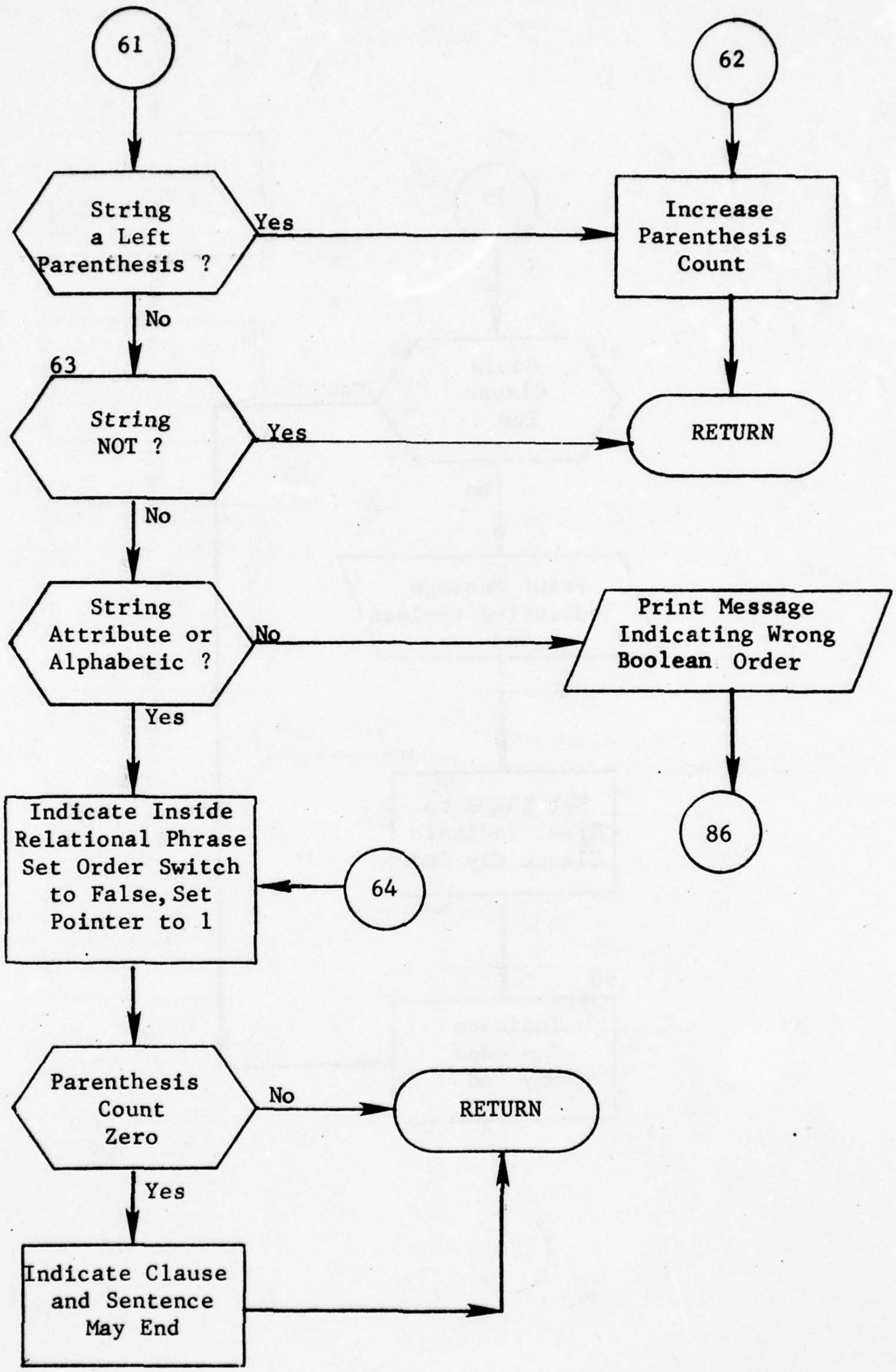



Figure 30. (Part 16 of 22)

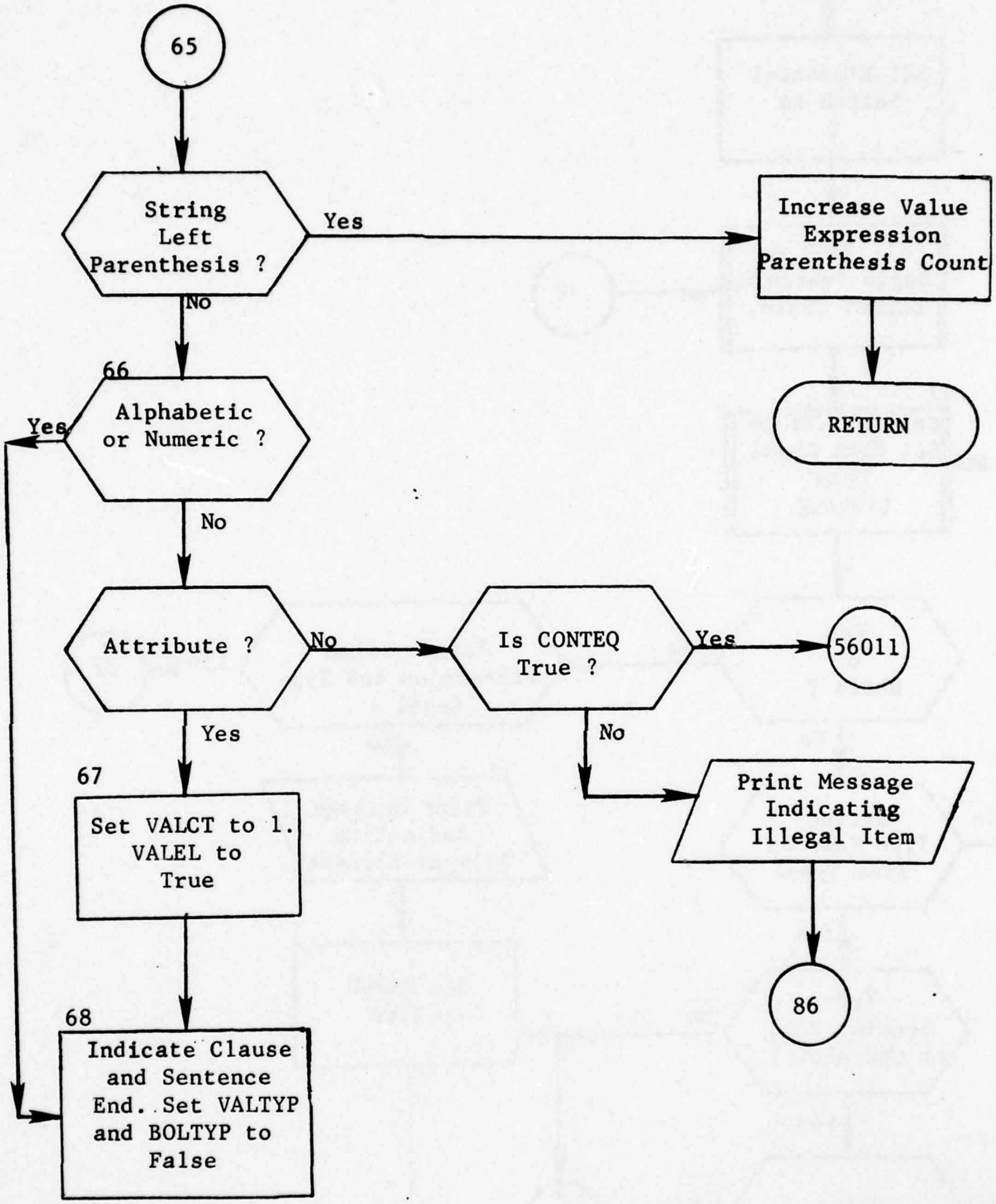



Figure 30. (Part 17 of 22)

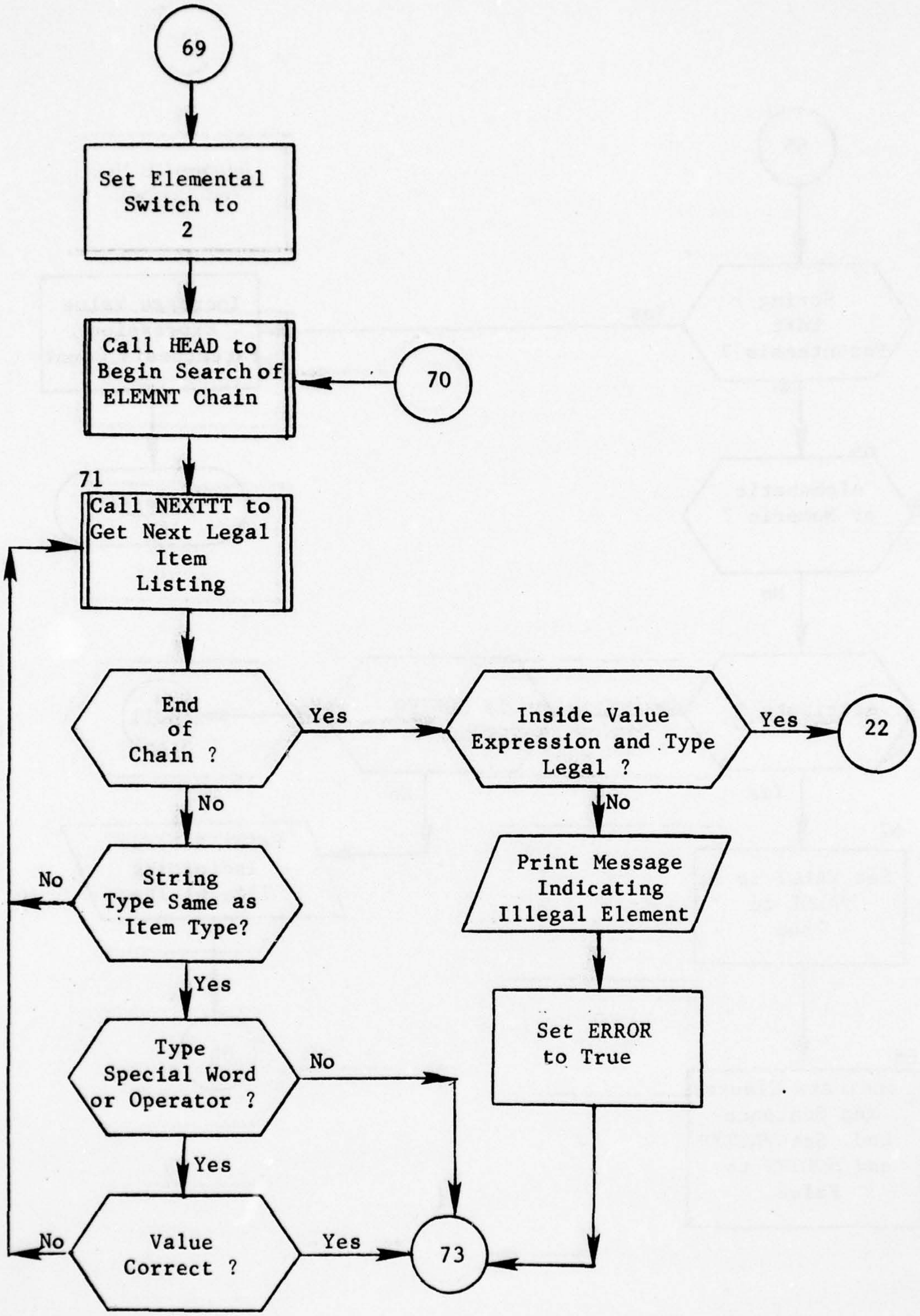



Figure 30. (Part 18 of 22)

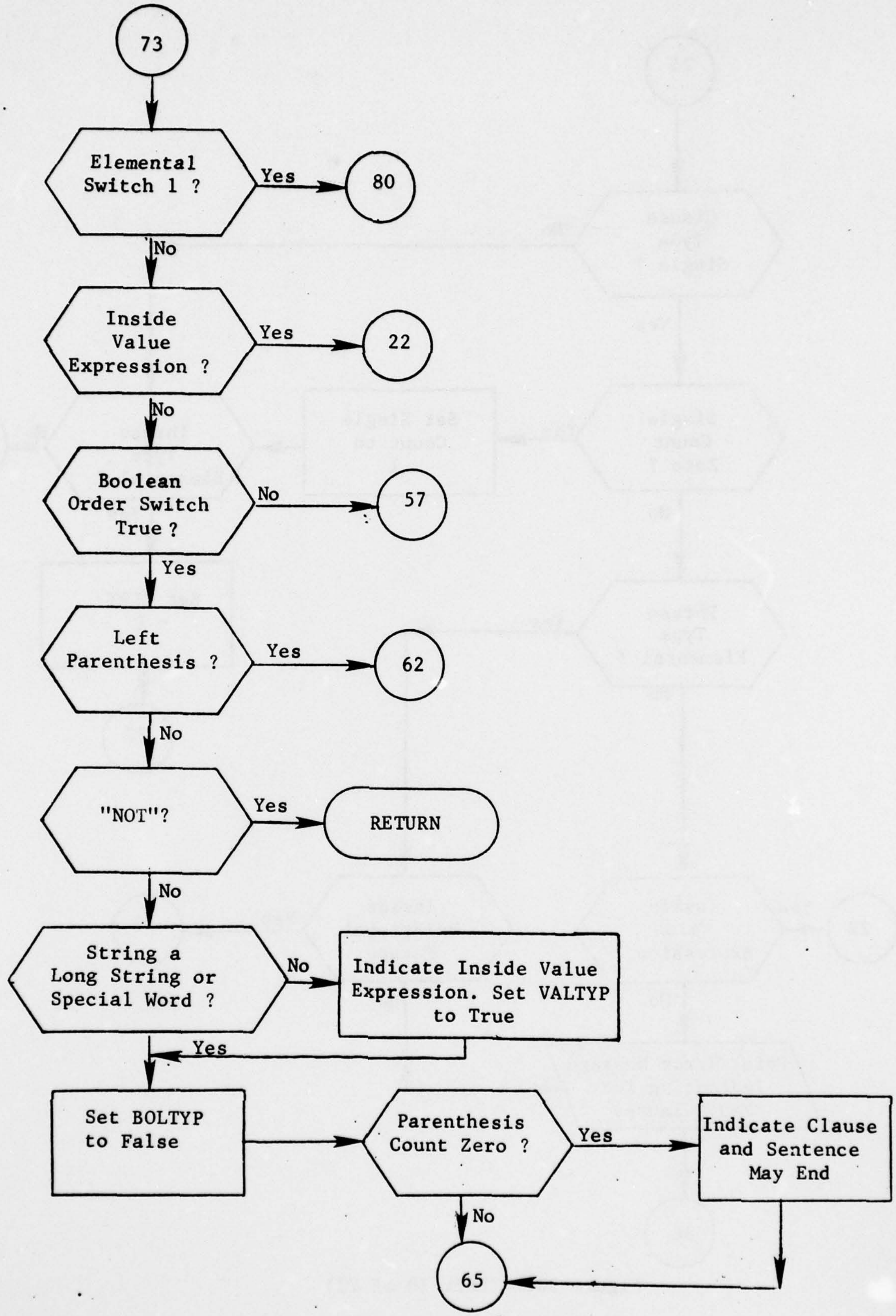



Figure 30. (Part 19 of 22)

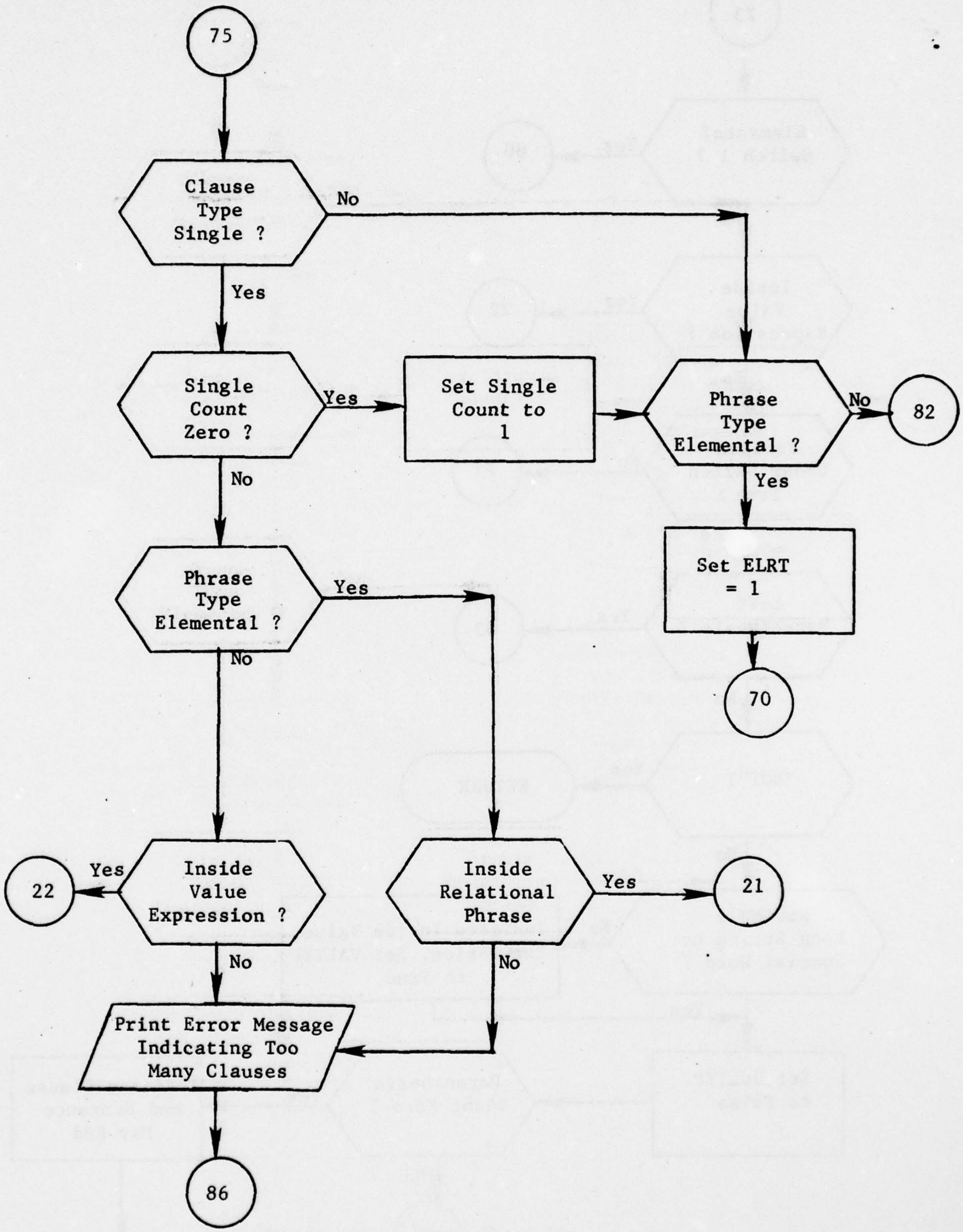



Figure 30. (Part 20 of 22)

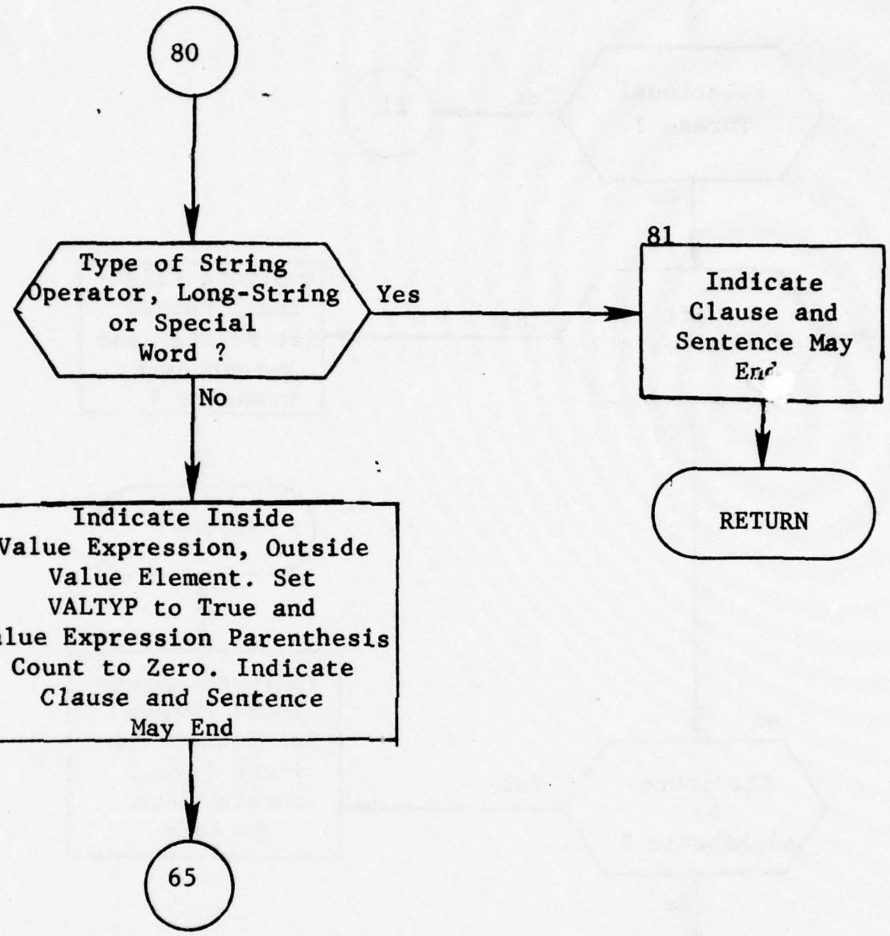



Figure 30. (Part 21 of 22)

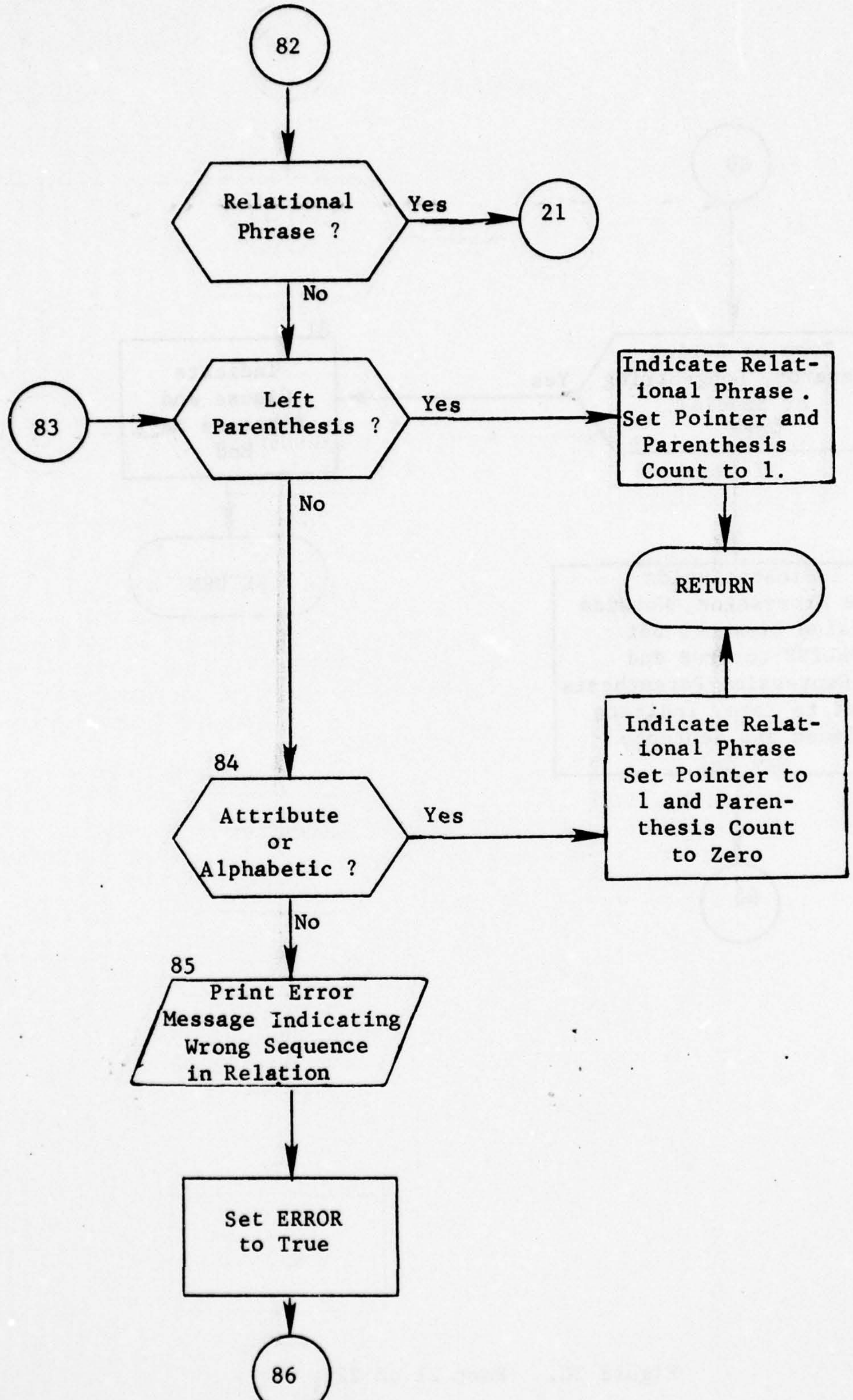



Figure 30. (Part 22 of 22)

is set to false and the second value processed. The EQUALS phrase may have a continuation. If the string is an AND (OR) and the clause is not boolian, a continuation is taking place. If the clause is boolian, a continuation is taking place. If the clause is boolean the CONTEQ switch is turned on so that later processing will know that the AND (OR) may be an EQUALS continuation rather than a boolean operator.

Statement 70 (figure 30)

Here elements are checked against the syntax directory. The ELEMNT chain is search for a match on type and (if the type is operator (type=1) or special word (type=5)) value. Since value expressions are allowed, this possibility is also checked. Finally the process returns to either the boolean or non-boolean code (ELRT).

3.9.3 Subroutine TABINS

PURPOSE: Creates ERRFND tables

ENTRY POINTS: TABINS

FORMAL PARAMETERS: TABTYP: Type of table

COMMON BLOCKS: C40, OOPS, STRING, SYMBOL, TABLZ

SUBROUTINES CALLED: RETRV, STORE

CALLED BY: ERRFND, LNGSTR

Method:

The process is somewhat different depending upon the value of TABTYP.

Numeric (TABTYP=1)

First the current numeric table is searched for a match. If it is not found any old tables are retrieved and searched for a match. If a match is found, the index of the match is packed into the symbol. If no match the current table is checked to see if it is full. If full it is added to the old tables and a new current table begun. Finally, the next numeric value is added to the current table and its index packed into the symbol.

Attribute (TABTYP=2)

The process here is much the same as for a numeric except that the attribute tables are used.

Alphabetic (TABTYP=3)

The process here is much the same as for a numeric except that the alphabetic tables are used. Each alphabetic constant occupies two words, thus the alphabetic tables contain only 50 entries.

Symbol (TABTYP=4)

If current table is full it is added to the list of old tables. Then the symbol is stored in the current table.

Subroutine TABINS is illustrated in figure 31.

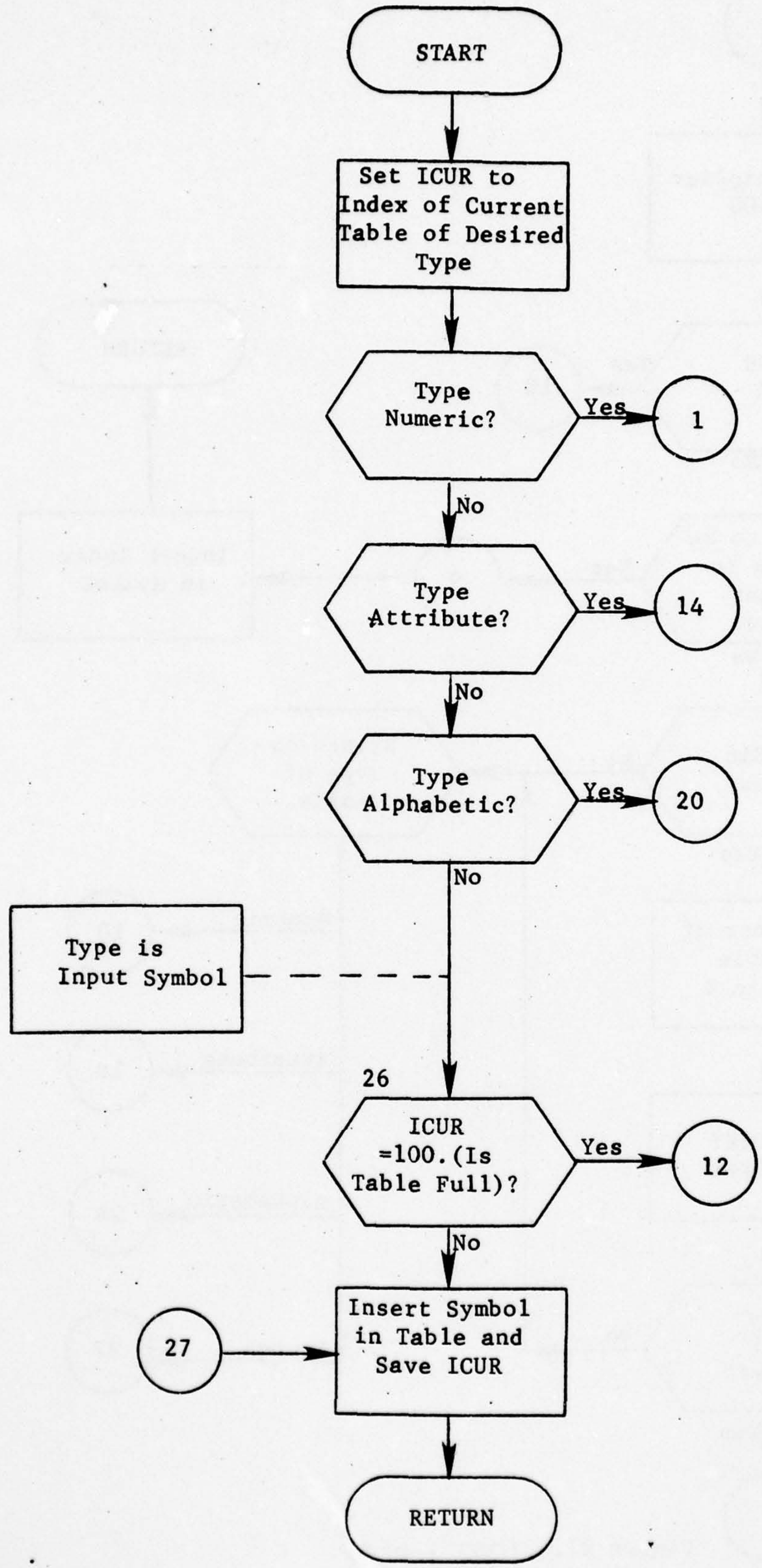



Figure 31. Subroutine TABINS (Part 1 of 6)

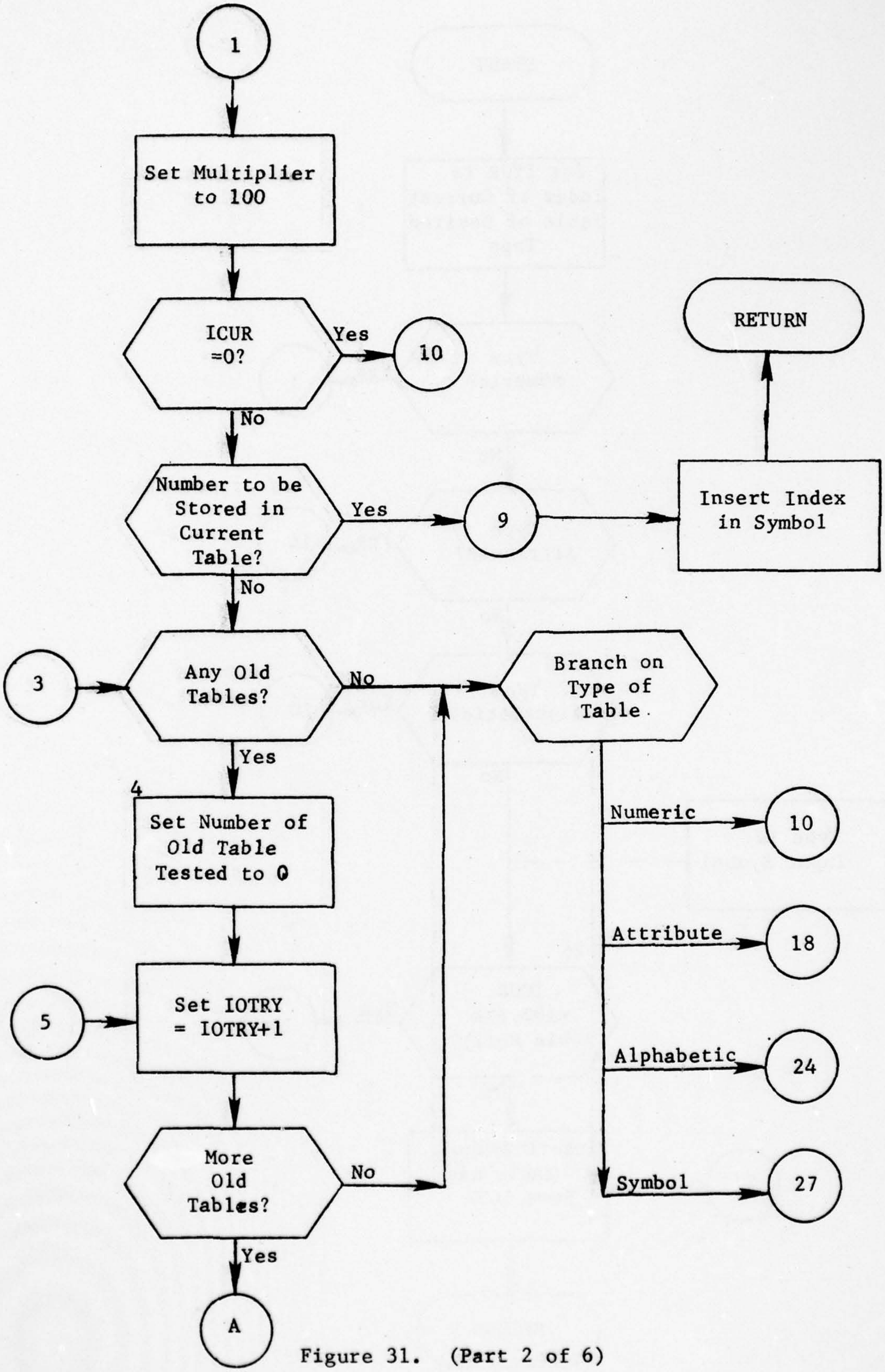



Figure 31. (Part 2 of 6)

AD-A054 377 COMMAND AND CONTROL TECHNICAL CENTER WASHINGTON D C F/G 15/7
THE CCTC QUICK-REACTING GENERAL WAR GAMING SYSTEM. (QUICK). PRO--ETC(U)
JUN 77 D J SANDERS, P F MAYKRANTZ, J M HERRIN
UNCLASSIFIED CCTC-CSM-MM-9-77-V1-PT-1 SBIE-AD-E100 051 NL

3 OF 5
AD
A054377

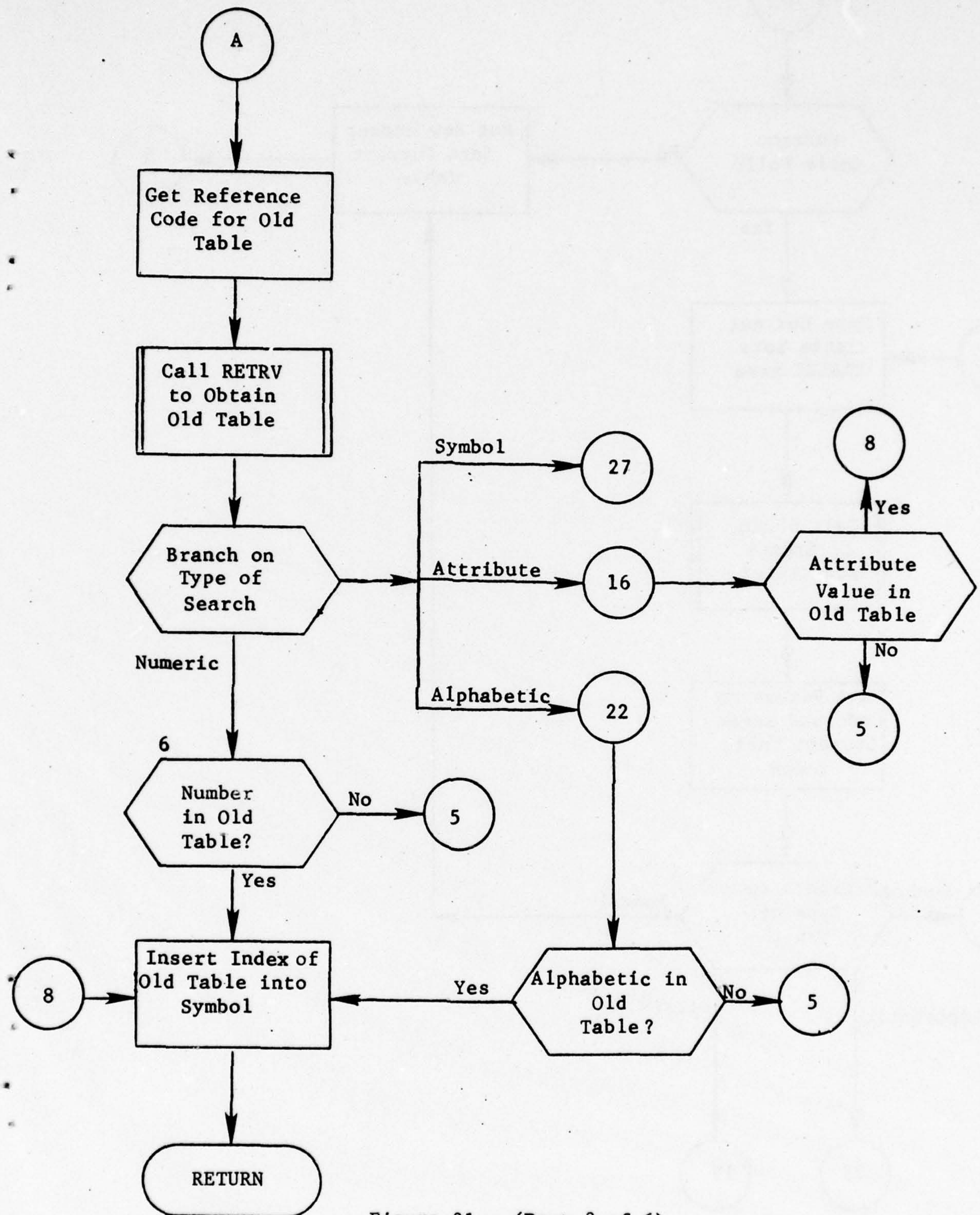



Figure 31. (Part 3 of 6)

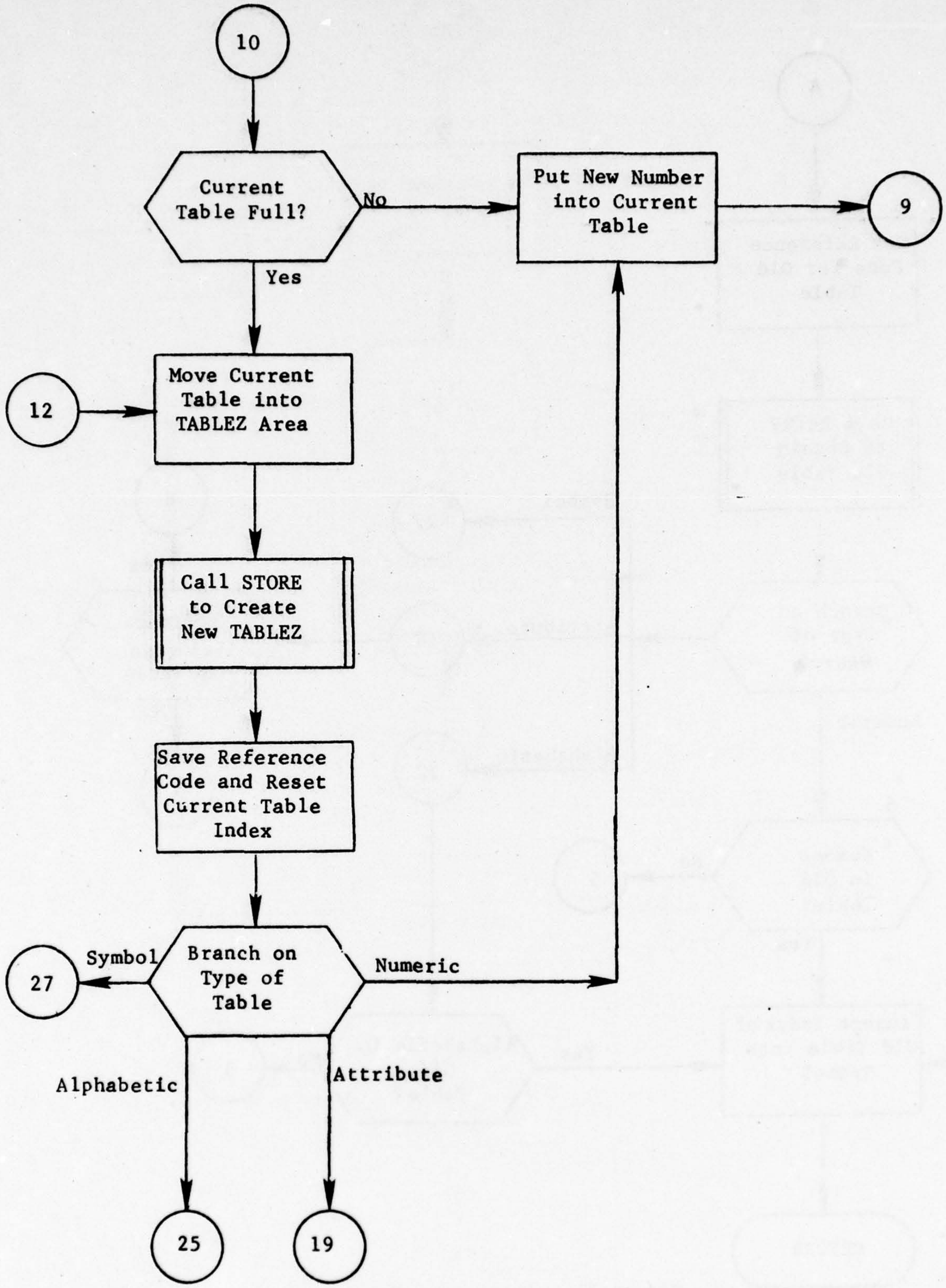



Figure 31. (Part 4 of 6)

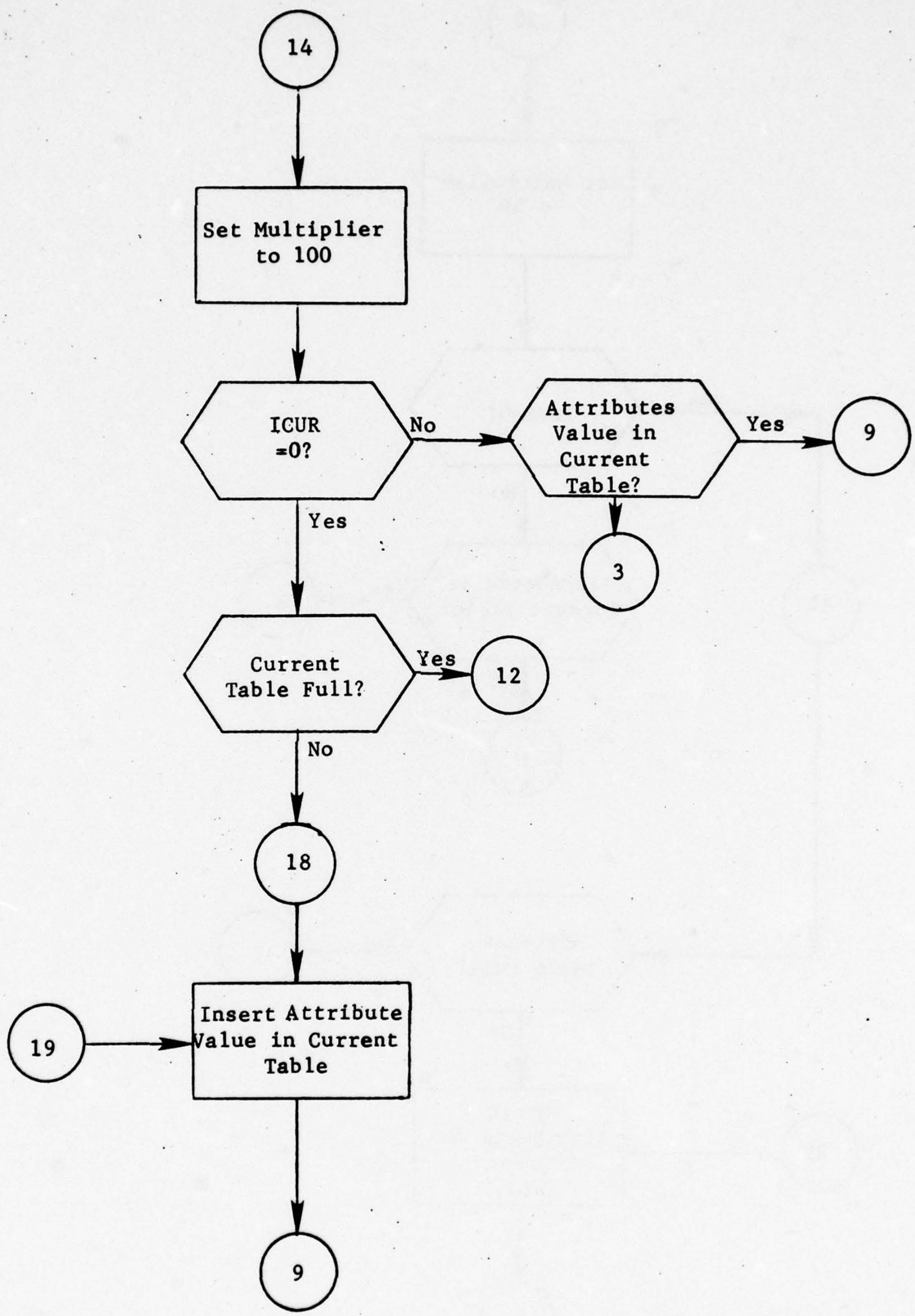



Figure 31. (Part 5 of 6)

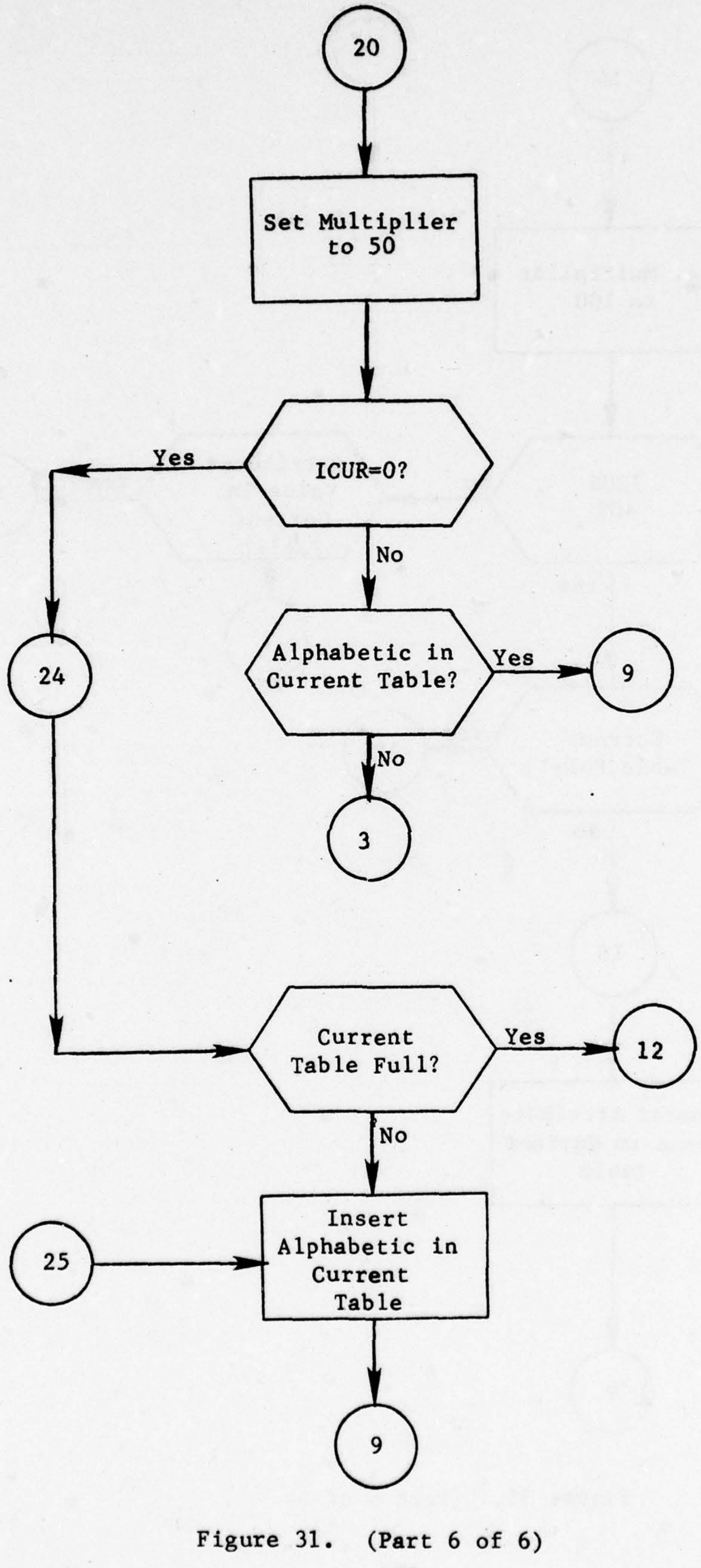



Figure 31. (Part 6 of 6)

### 3.9.4 Subroutine WEBSTR

| | |
|---|---|
| PURPOSE: | Looks up input strings in dictionary |
| ENTRY POINTS: | WEBSTR |
| FORMAL PARAMETERS: | None |
| COMMON BLOCKS: | C10, C15, C30, STRING |
| SUBROUTINES CALLED: | HDFND, NEXTTT, RETRV |
| CALLED BY: | ERRFND, INICOP |

Method:

First the tab character is created from the first two characters of ALPHA (from common block STRING). ISWT is set to assure a complete search of the TAB chain. The TAB chain is thus searched for the tab character. If the tab is not found, the process ends. If it is found, the WØRD chain is searched for a match for ALPHA. If a match is found, TXPE and VALUE are set.

Subroutine WEBSTR is illustrated in figure 32.

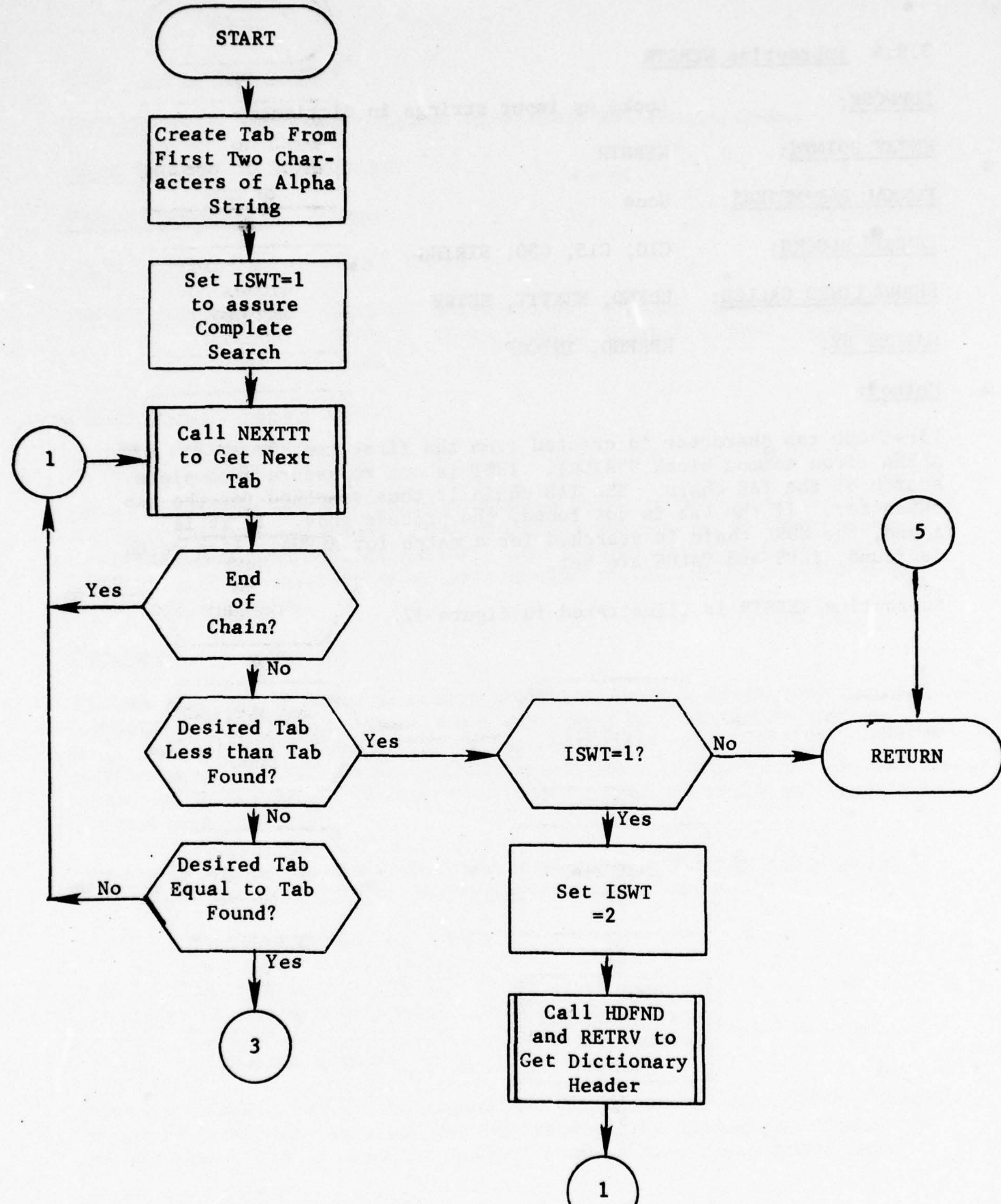



Figure 32. Subroutine WEBSTR (Part 1 of 2)

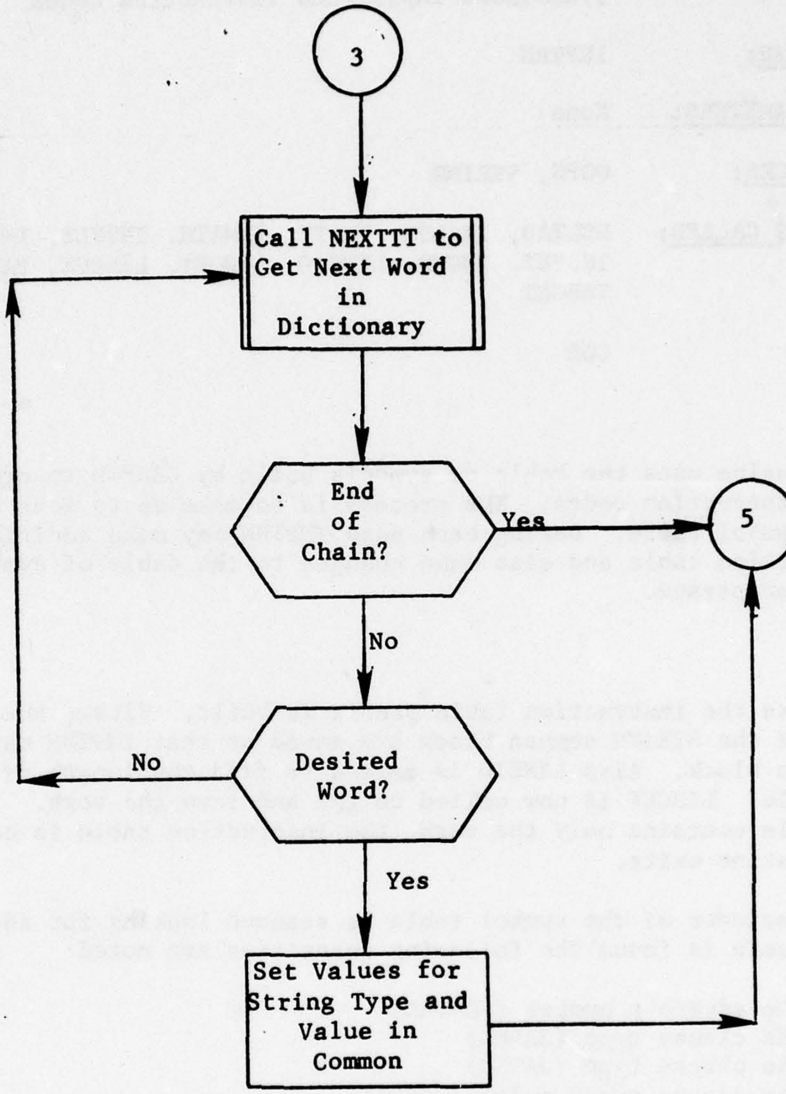



Figure 32. (Part 2 of 2)

## 3.10 Subroutine INPTRN*

PURPOSE: Translates input into instruction codes

ENTRY POINTS: INPTRN

FORMAL PARAMETERS: None

COMMON BLOCKS: OOPS, STRING

SUBROUTINES CALLED: DELTAB, INALPH, INATT, INMATH, INSFLS, INSNIT, INSPUT, INUMB, LINEIO, LINGET, LINPUT, PARLEV, TARGET

CALLED BY: COP

Method:

This subroutine uses the table of symbols built by ERRFND to create the table of instruction codes. The process is to make up to four passes over the symbol table. During each pass INPTRN may make additions to the instruction table and also make changes to the table of symbols to assist later passes.

Pass One

In this pass the instruction table prefix is built. First, the current contents of the STRING common block are saved so that INPTRN may use this common block. Also LINEIO is called to find the length of the symbol table. LINGET is now called to get and save the verb. If the symbol table contains only the verb, the instruction table is completed and the routine exits.

Now the remainder of the symbol table is scanned looking for adverbs. When an adverb is found the following quantities are noted:

- o The adverb's number (JADVAL)
- o The clause type (JADEL)
- o The phrase type (JADEX)
- o The clause start point (JADST)
- o The clause end point (JADEND) (This item is found for the previous adverb)

After an adverb is found the symbol for it is zeroed out. If the adverb phrase type is not elemental the remainder of the clause is scanned for collections. This is done by keeping track of left parentheses and

*Main routine of overlay INPT

when a comma is encountered, the last left parenthesis of the proper level and the next right parenthesis of the same level have their symbols changed. Left parentheses symbols are changed so that bits 30-35= 21, right parentheses symbols are changed so that bits 30-35=22.

Passes Two through Four are performed for each adverb in the order in which they were input. When all adverbs have been processed and the instruction table is complete, INSFLS is called to save the instruction table. DELTAB is called to delete ERRFND tables. The contents of the STRING common block are now restored and the subroutine exits.

Pass Two

The first step is to set up the pointer in the instruction prefix. If the adverb is null (JADEX=4) this is all that is done. A different process is used for adverbs with elemental phrases (JADEL=3). For these adverbs, each symbol of the clause in turn is converted to the appropriate "follower instruction."

If the adverb's phrases are relational phrases this pass is used to translate mathematical calculations and to replace any AND or OR operators which are, in reality, connectors for EQUALS and BETWEEN relations. In this process the branch RELOP is used to keep track of the part of the relational phrase expected next. Values for RELOP are:

RELOP = 1 - looking for an operator
RELOP = 2 - looking for a collector or object
RELOP = 3 - looking for object
RELOP = 4 - looking for continuation

When the beginning of the object is found, it is noted. While an object is being scanned the presence of any math operator is noted. When the end of the object is found, the processes branches depending upon whether any math operators were noted (MATH=true). If so, the code beginning at statement 55 (see figure 33) is used. If not, all parentheses in the object are removed.

The code at statement 55 is a process whereby each level of parentheses is resolved separately. The first step is to call PARLEV which flags each parenthesis with its level and notes the highest level. Then each level of parentheses is converted to instructions needed to calculate the value each expression within the parentheses. The series of instructions is terminated by the instruction to store the calculated value in an internal variable. The parenthetical expression in the symbol table is replaced by a single non-zero symbol indicating the index number of the internal variable (bits 30-35=25, bits 0-29=index). As each level is evaluated more of the expression is removed until finally only one non-zero symbol for the internal variable containing the result remains.

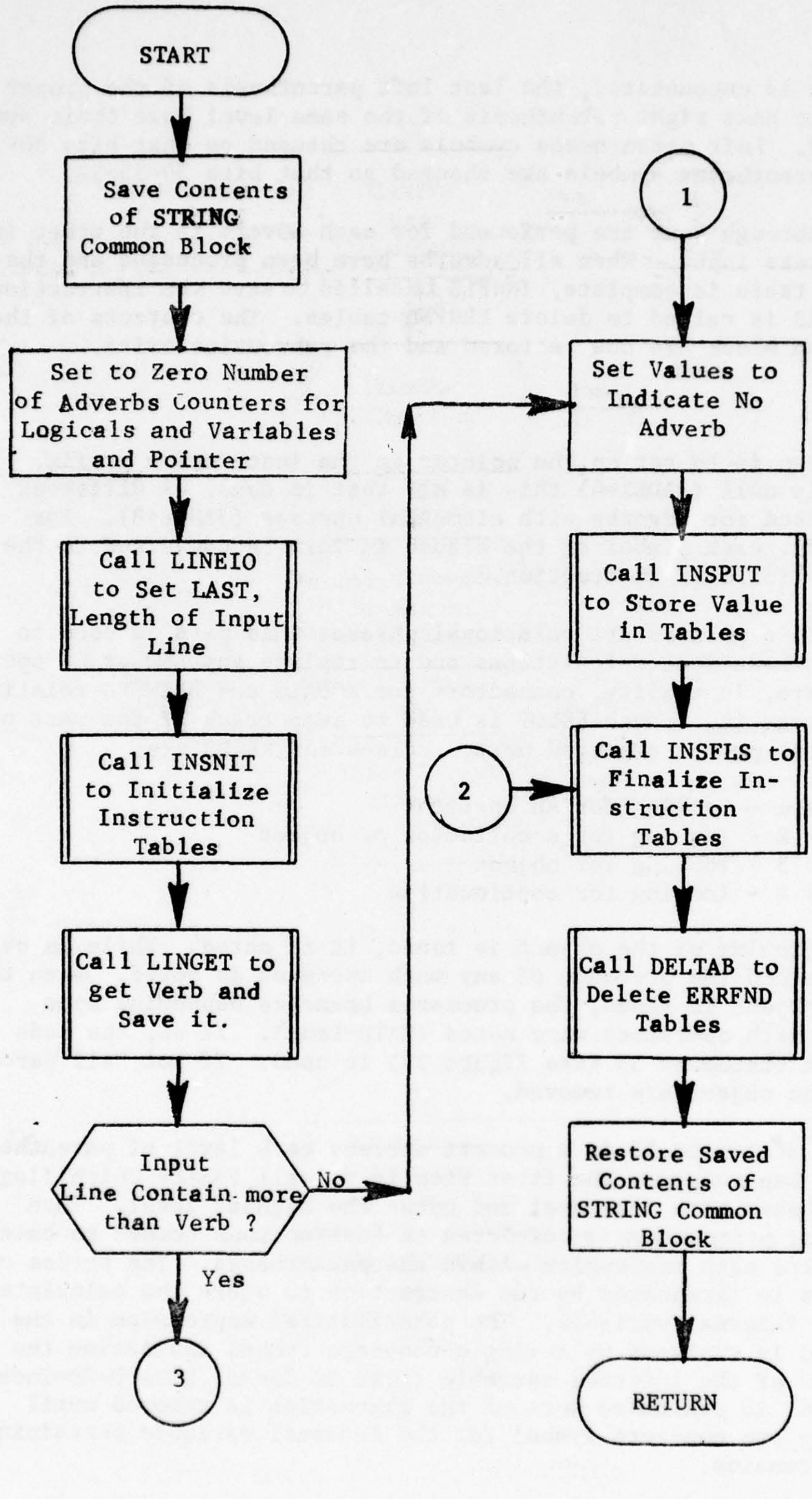



Figure 33. Subroutine INPTRN (Part 1 of 36)

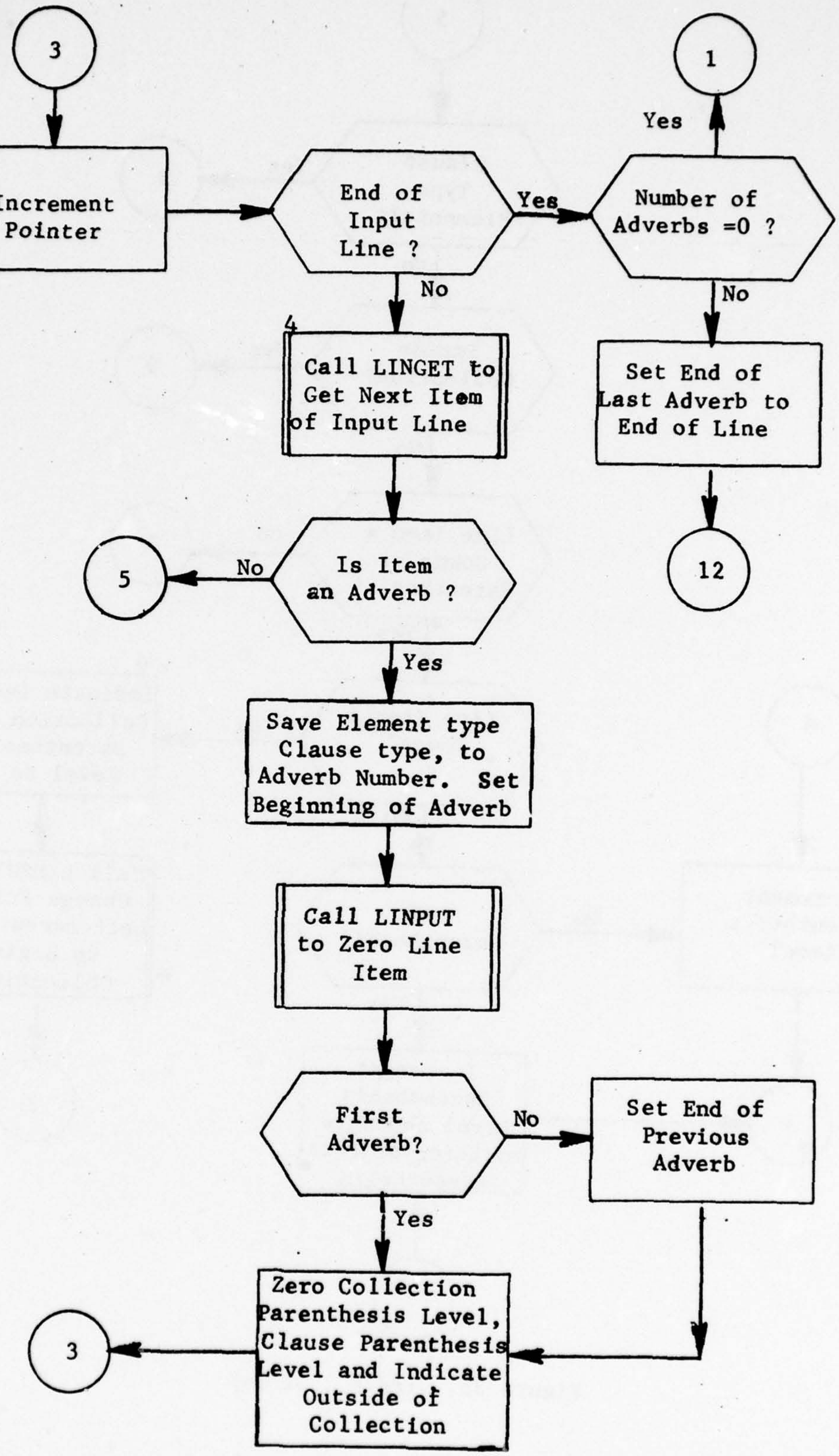



Figure 33. (Part 2 of 36)

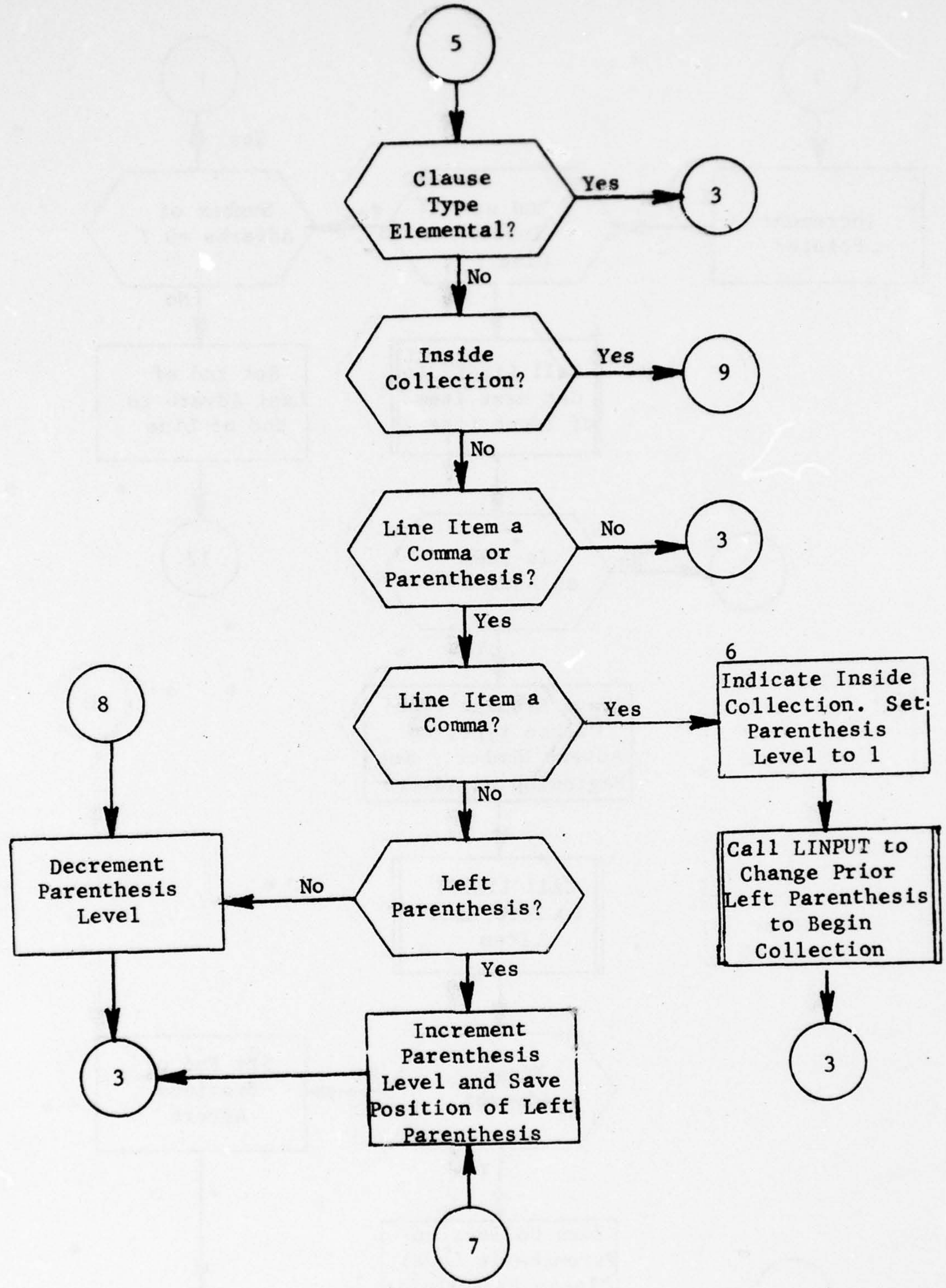



Figure 33. (Part 3 of 36)

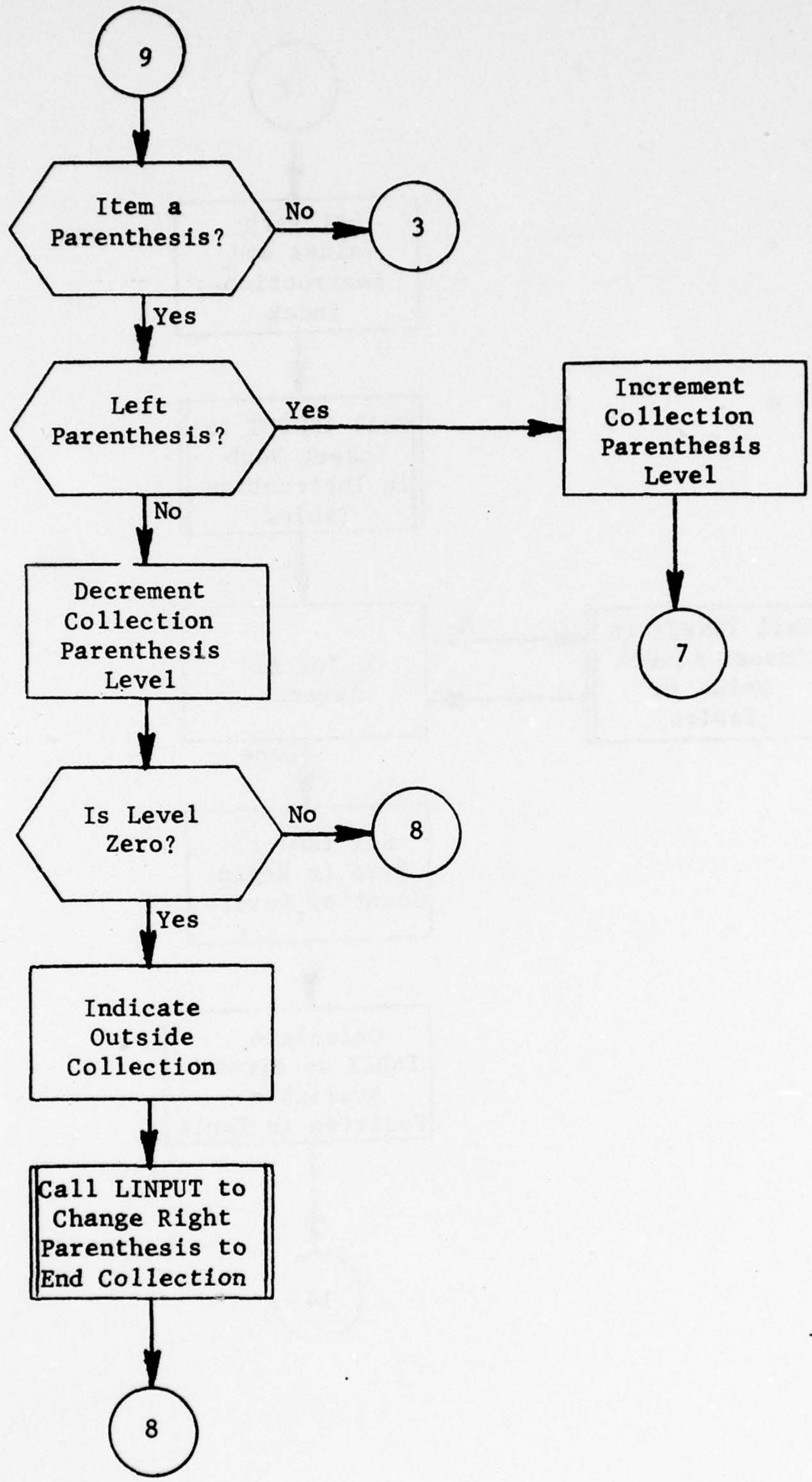



Figure 33. (Part 4 of 36)

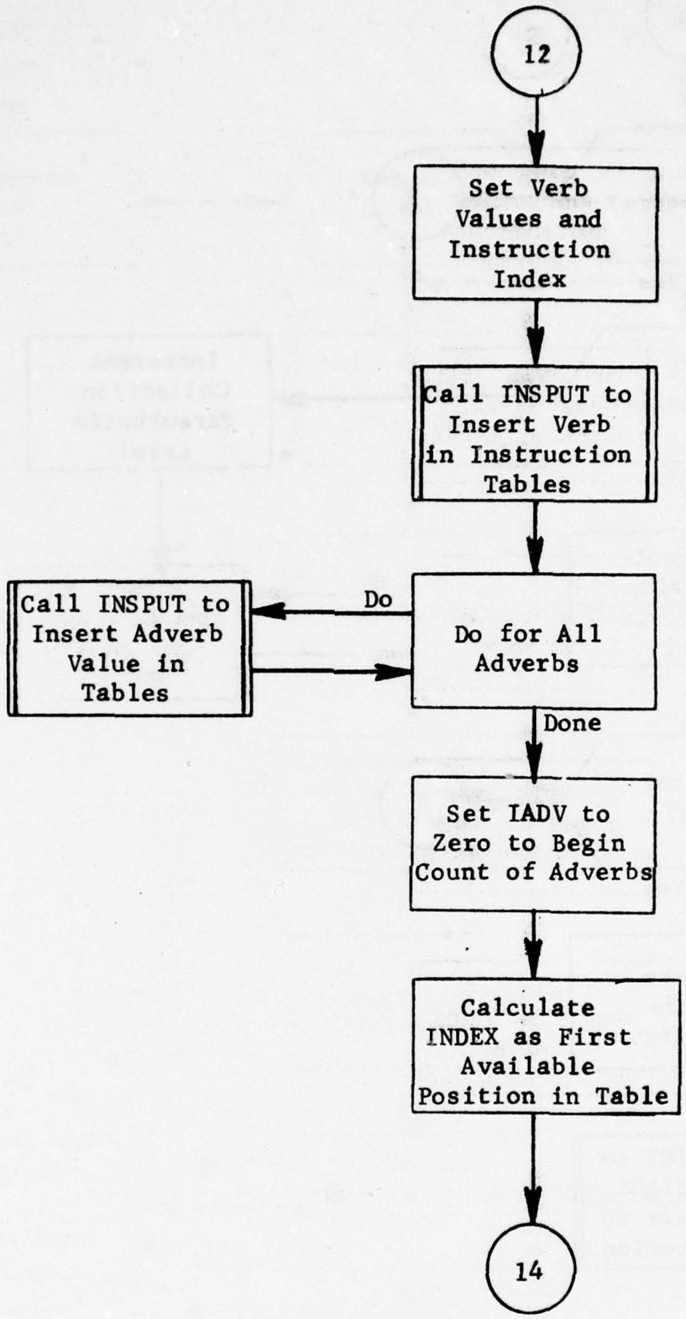



Figure 33. (Part 5 of 36)

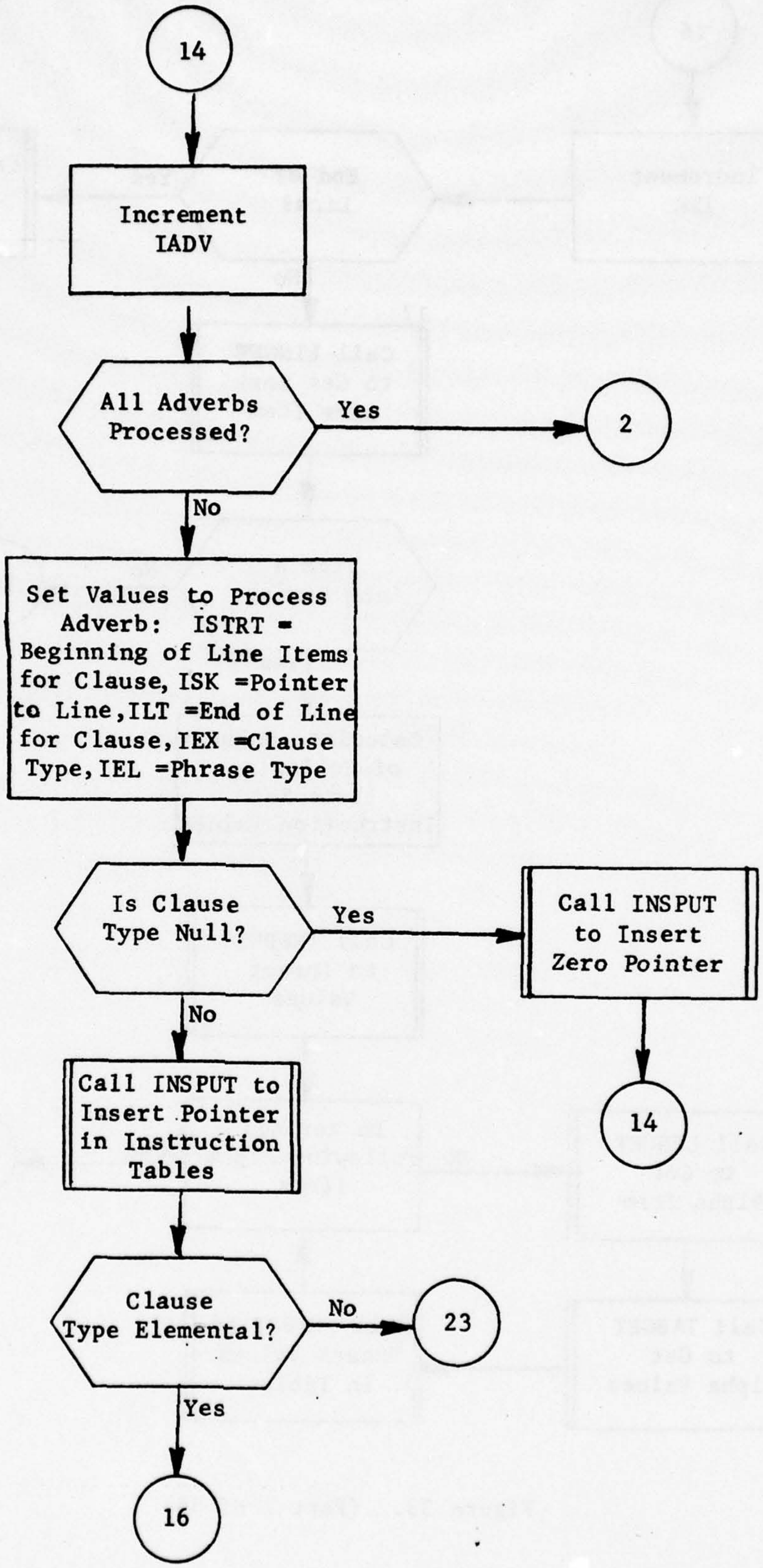



Figure 33. (Part 6 of 36)

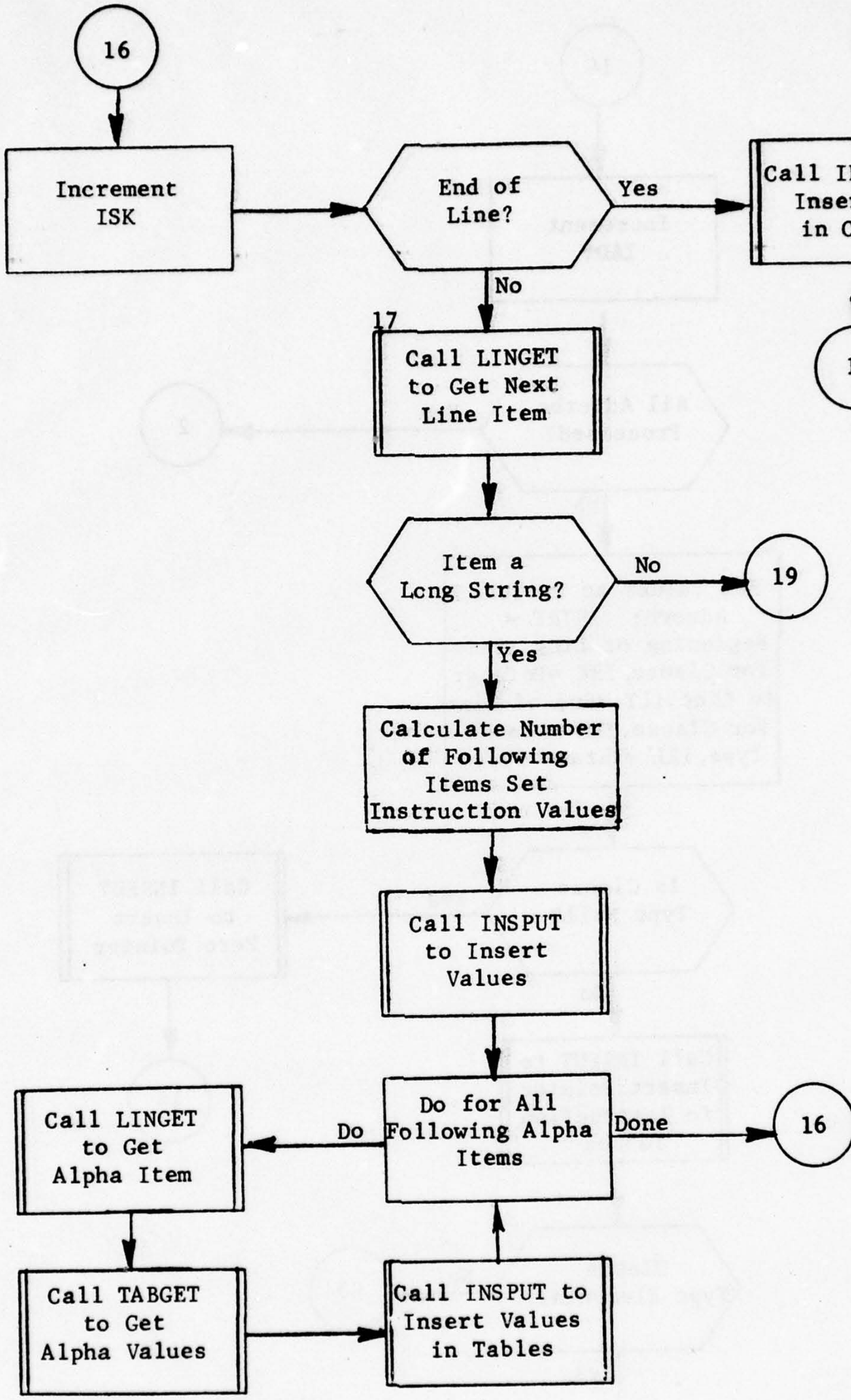



Figure 33. (Part 7 of 36)

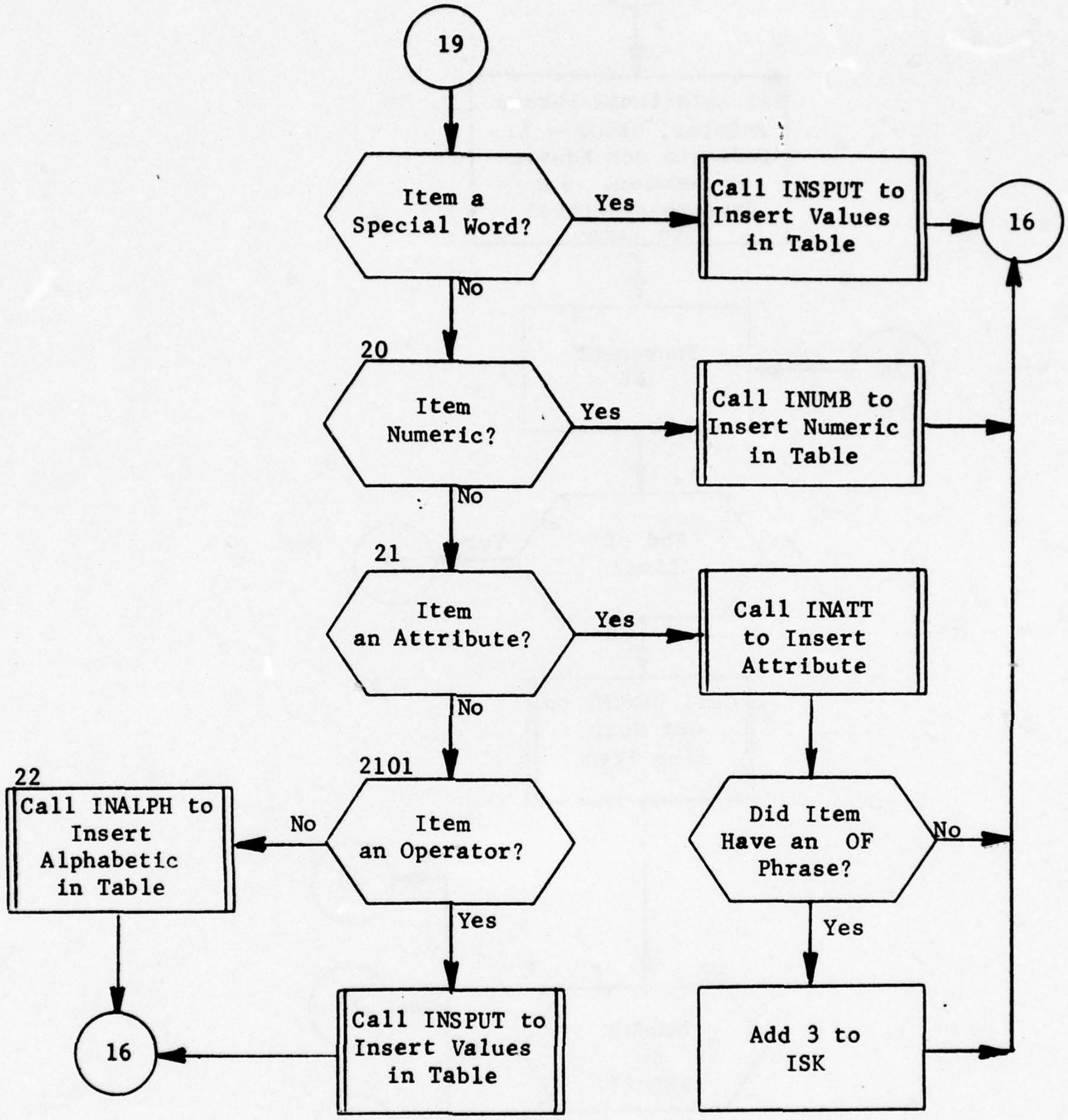



Figure 33. (Part 8 of 36)

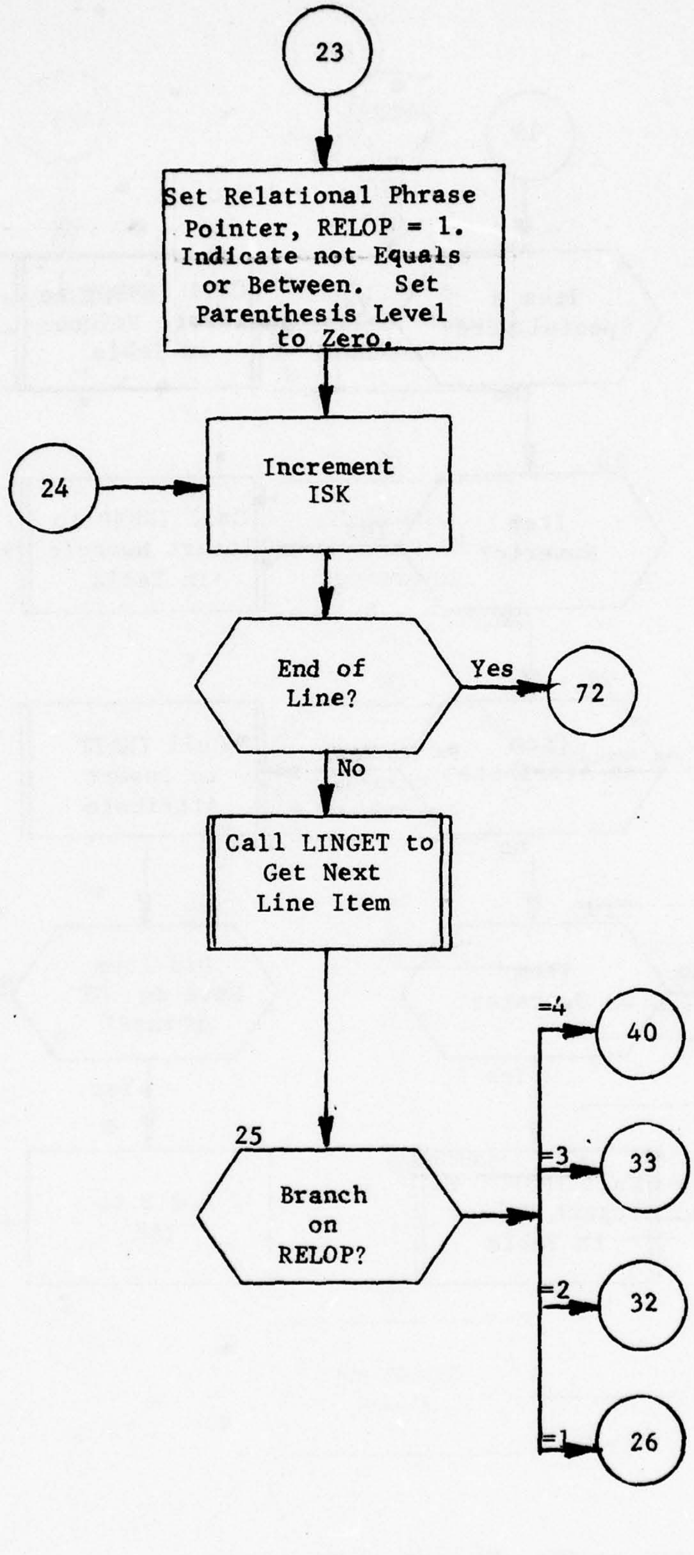



Figure 33. (Part 9 of 36)

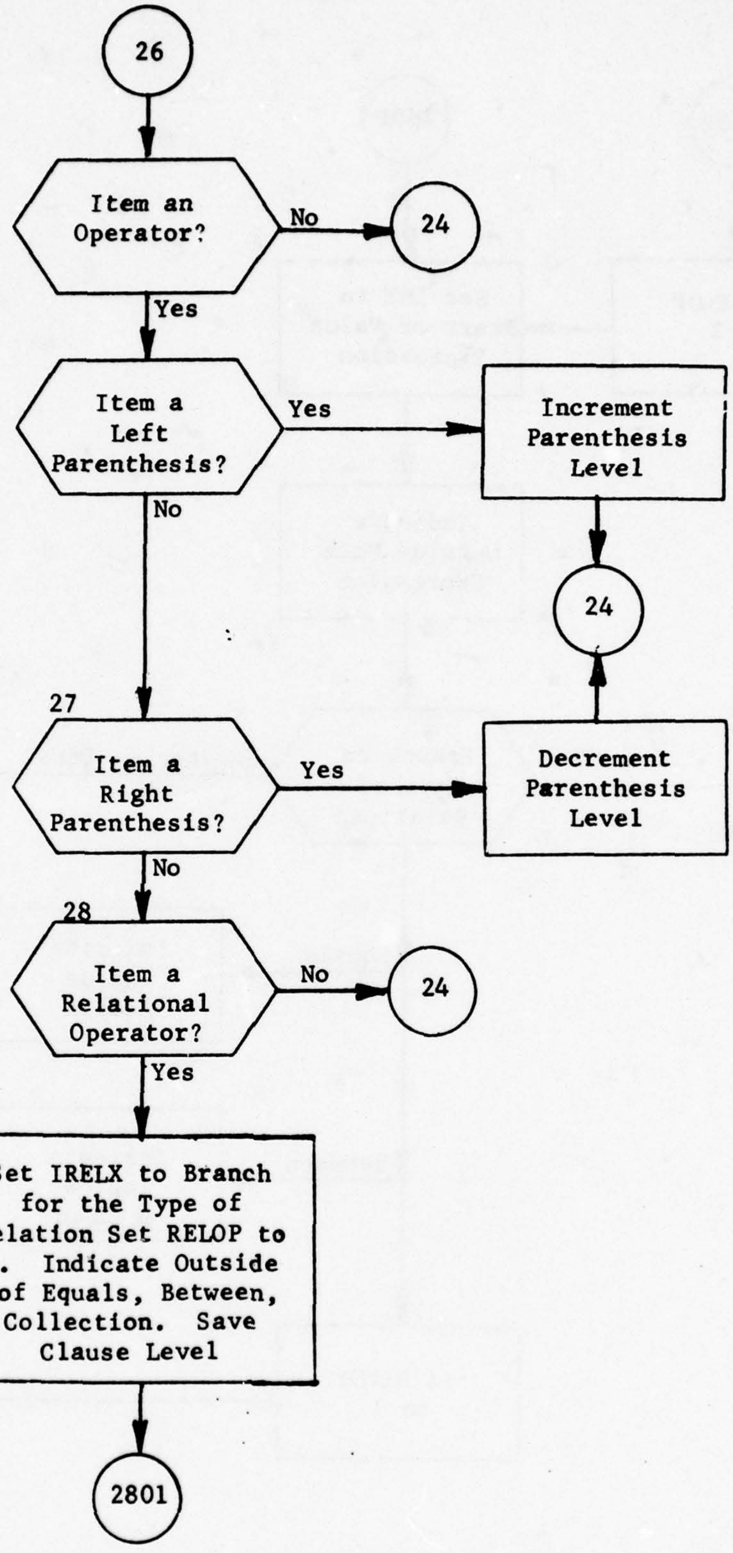



Figure 33. (Part 10 of 36).

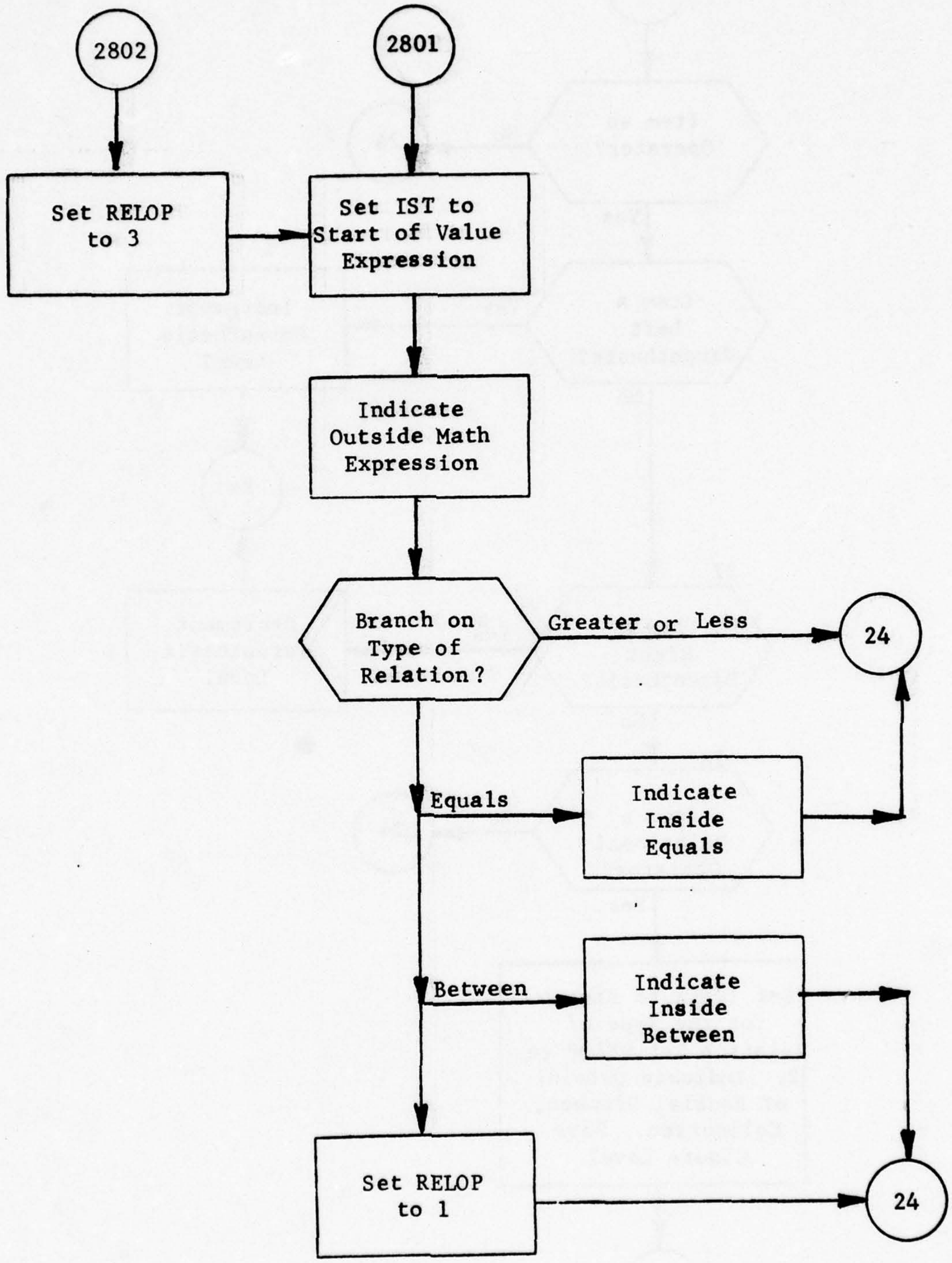



Figure 33. (Part 11 of 36)

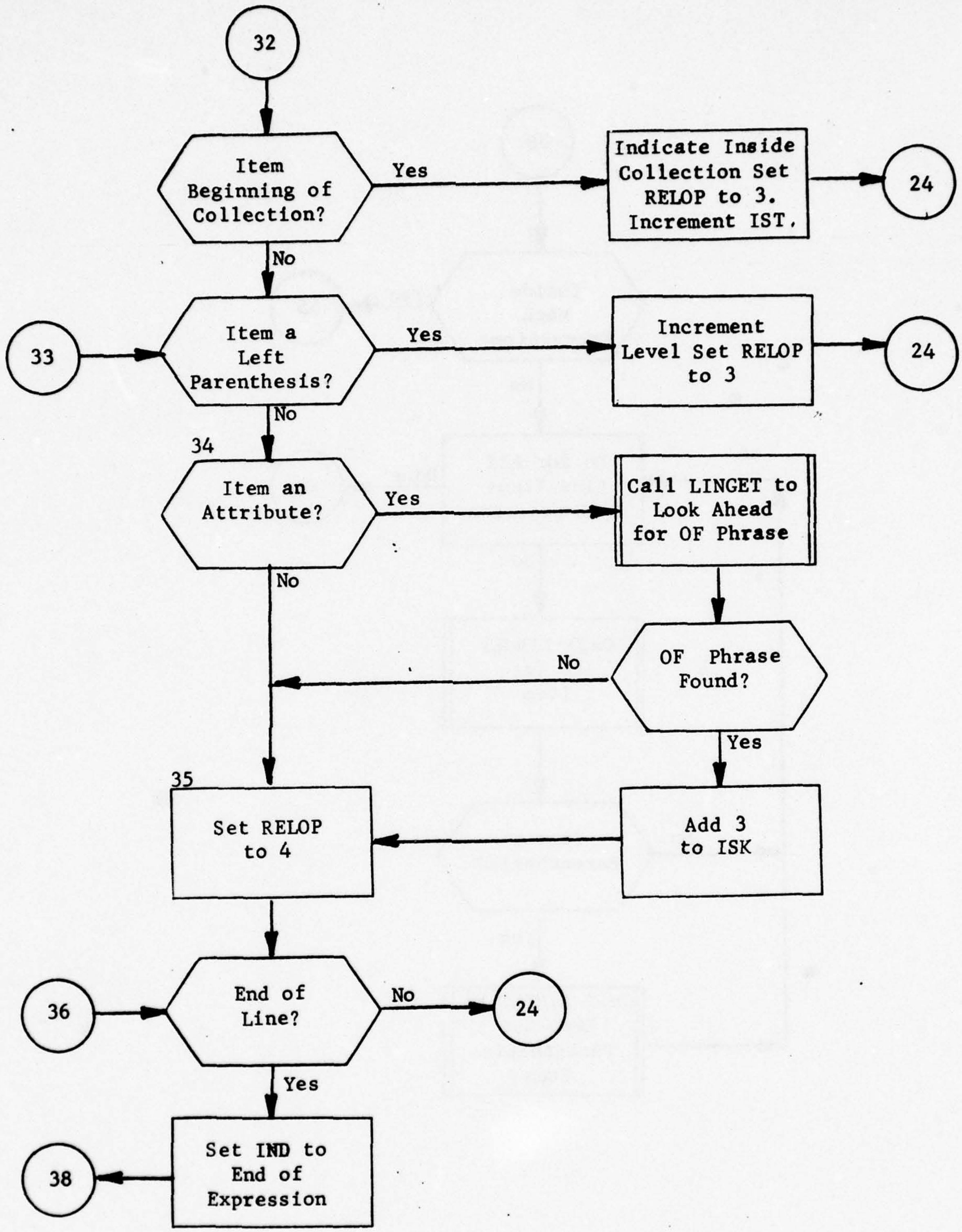



Figure 33. (Part 12 of 36)

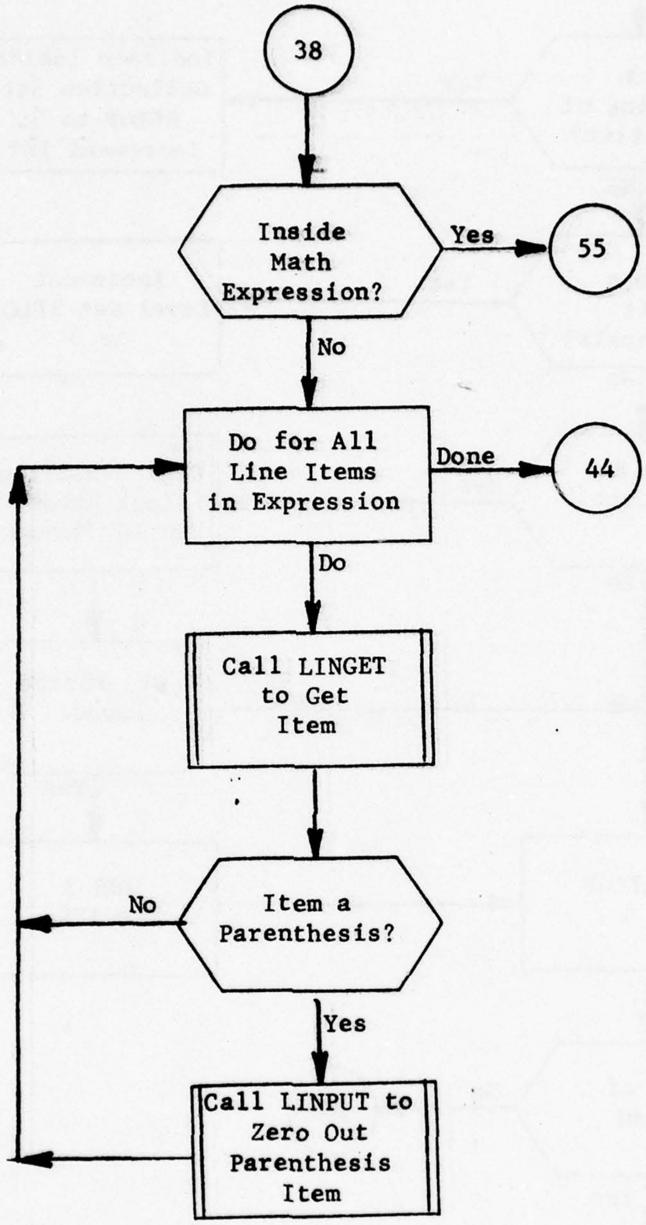



Figure 33. (Part 13 of 36)

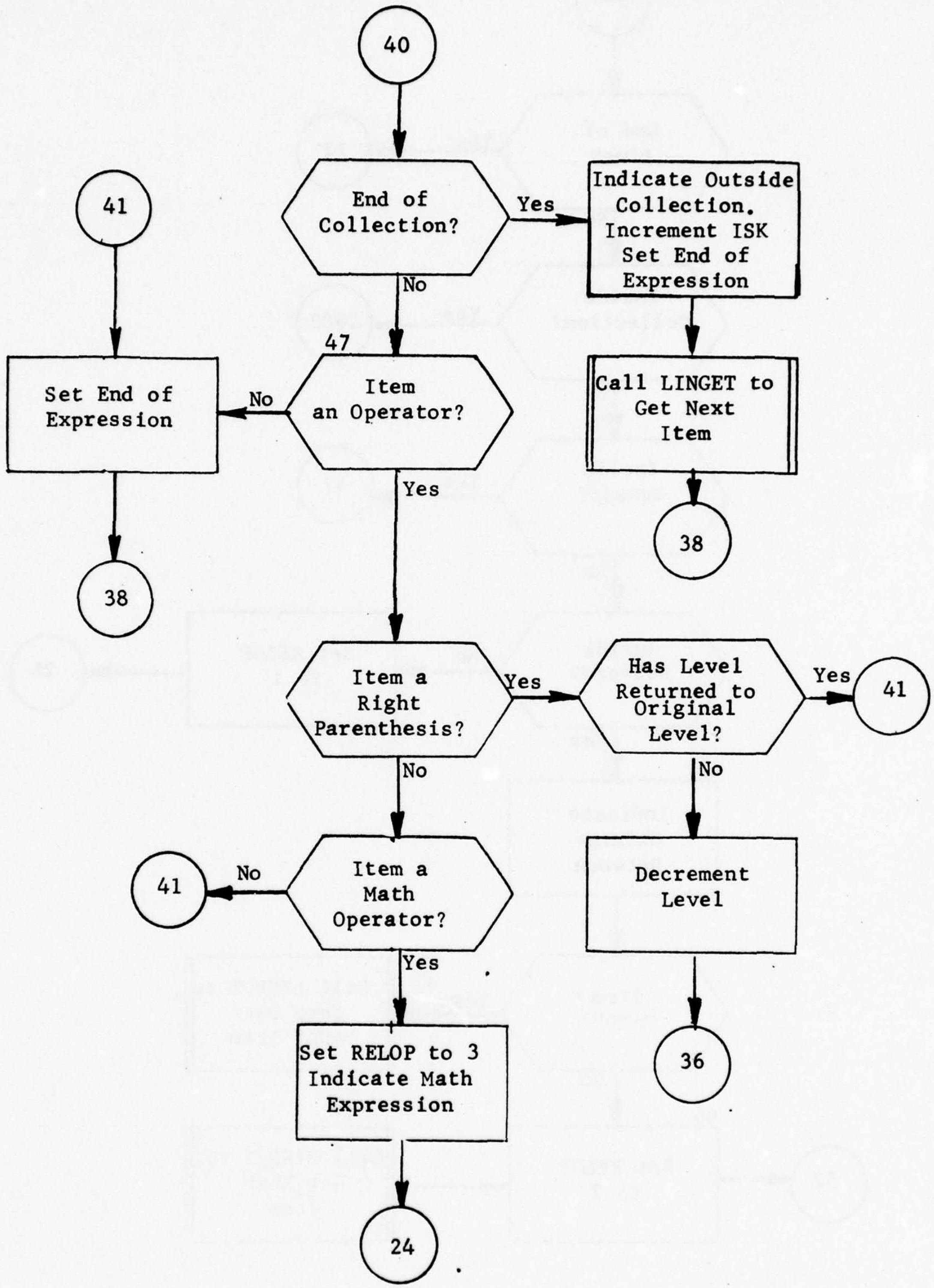



Figure 33. (Part 14 of 36)

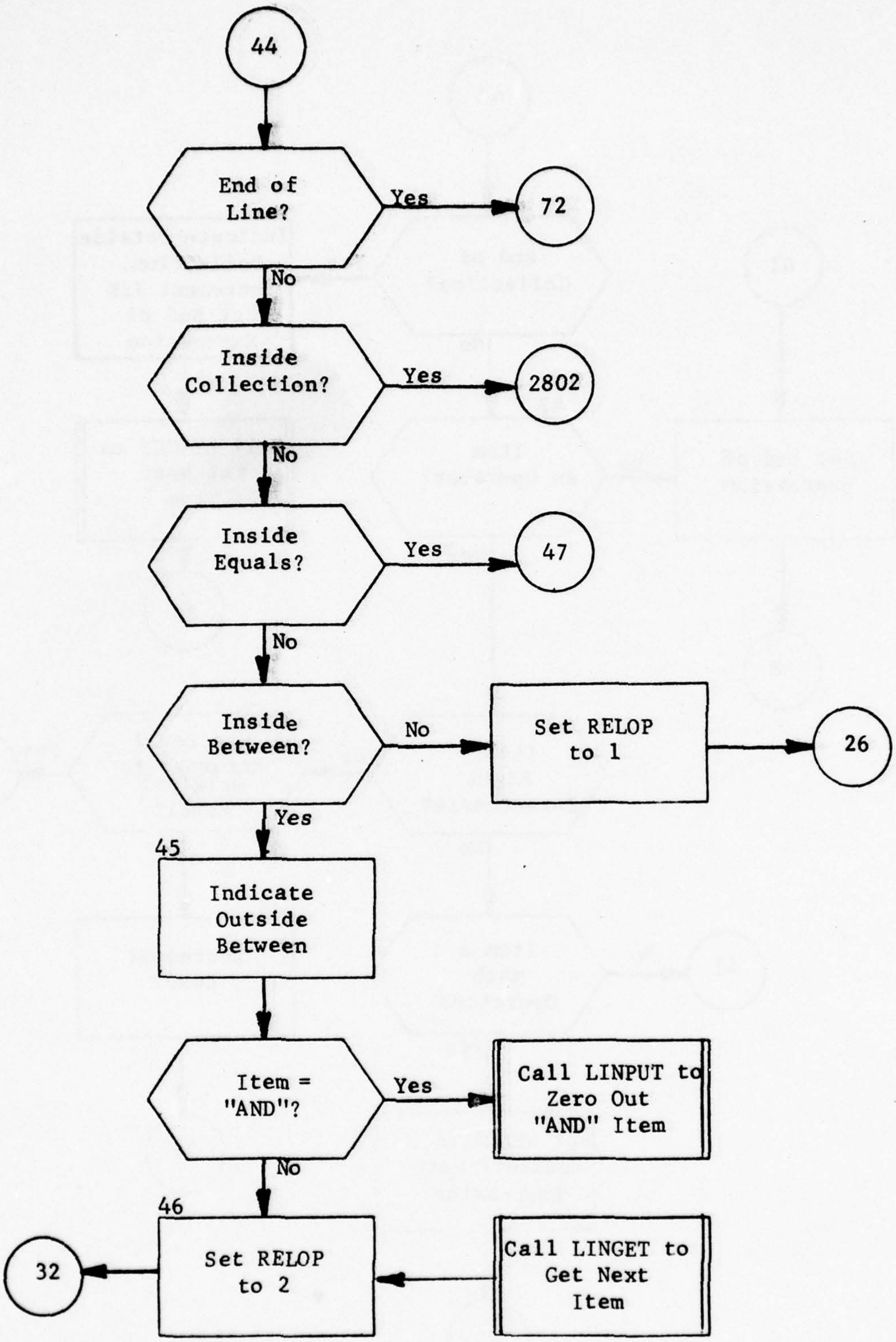



Figure 33. (Part 15 of 36)

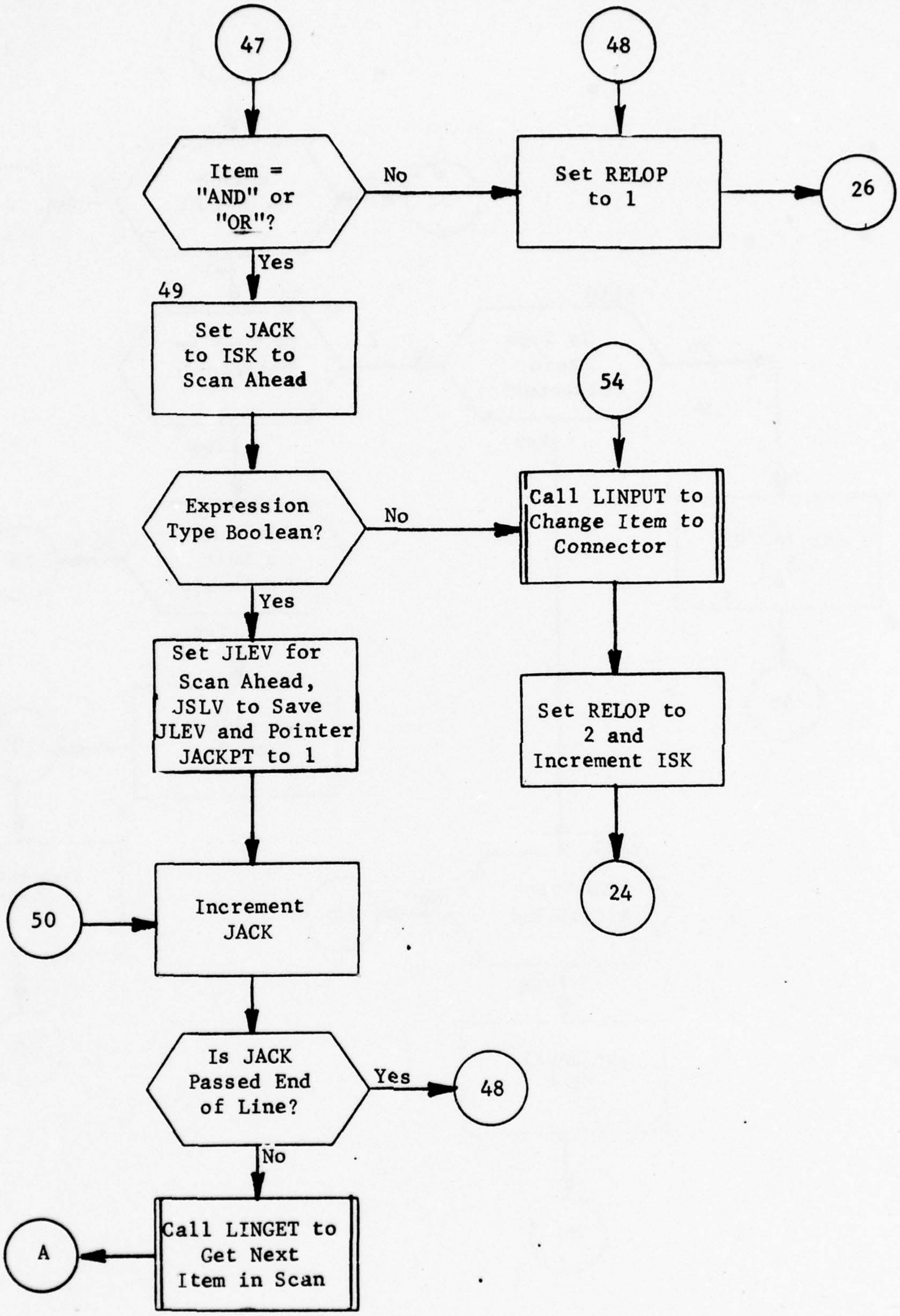



Figure 33. (Part 16 of 36)

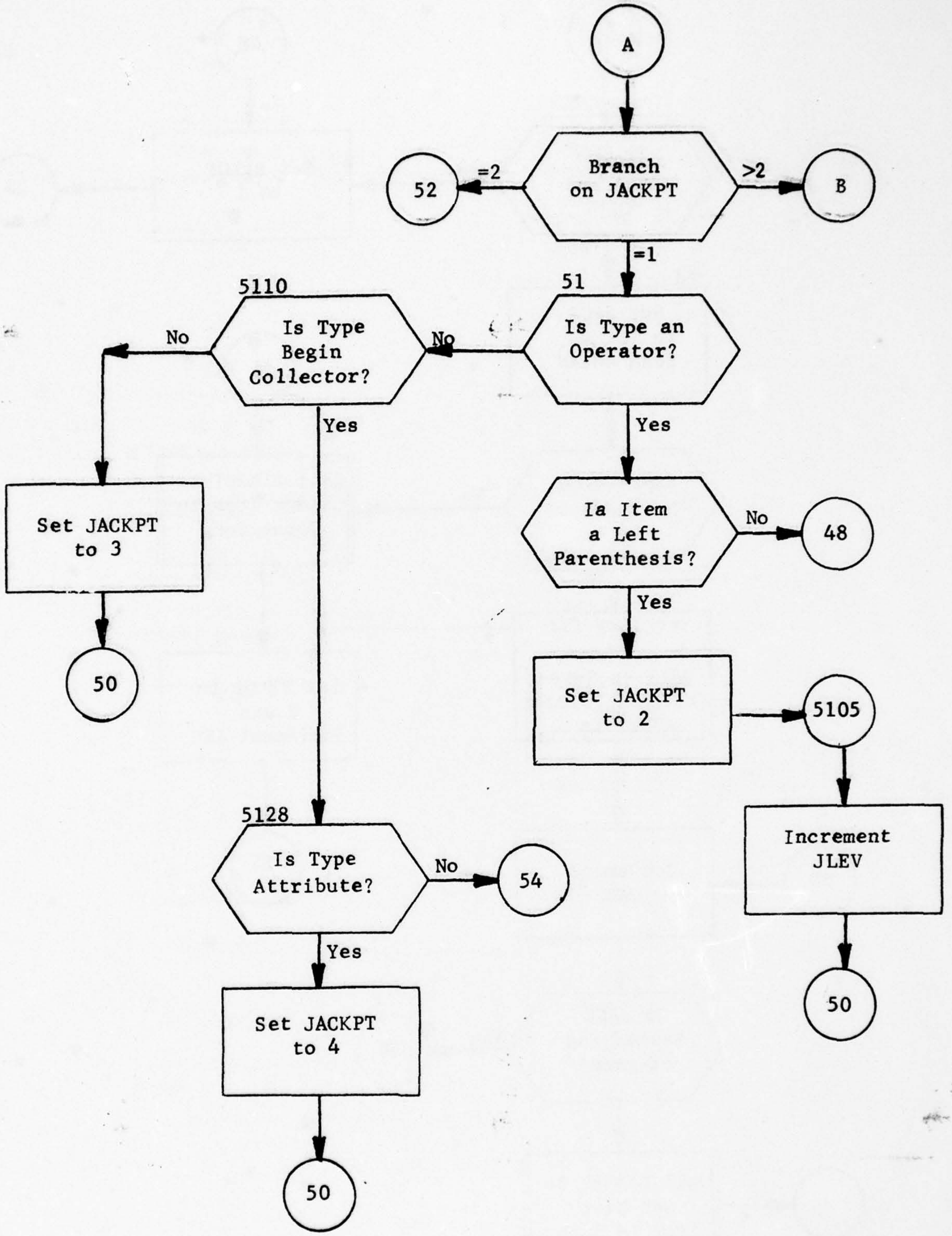



Figure 33. (Part 17 of 36)

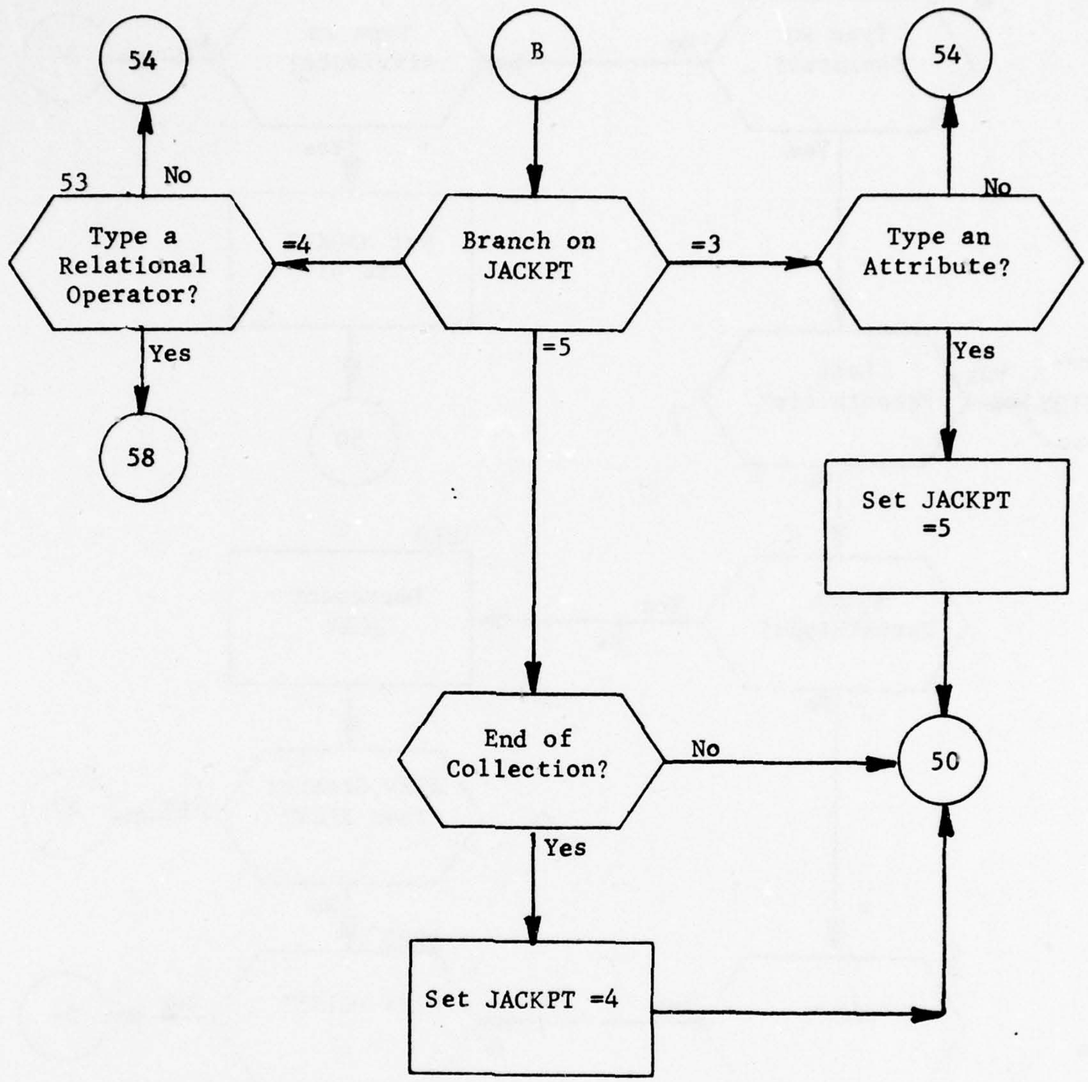



Figure 33. (Part 18 of 36)

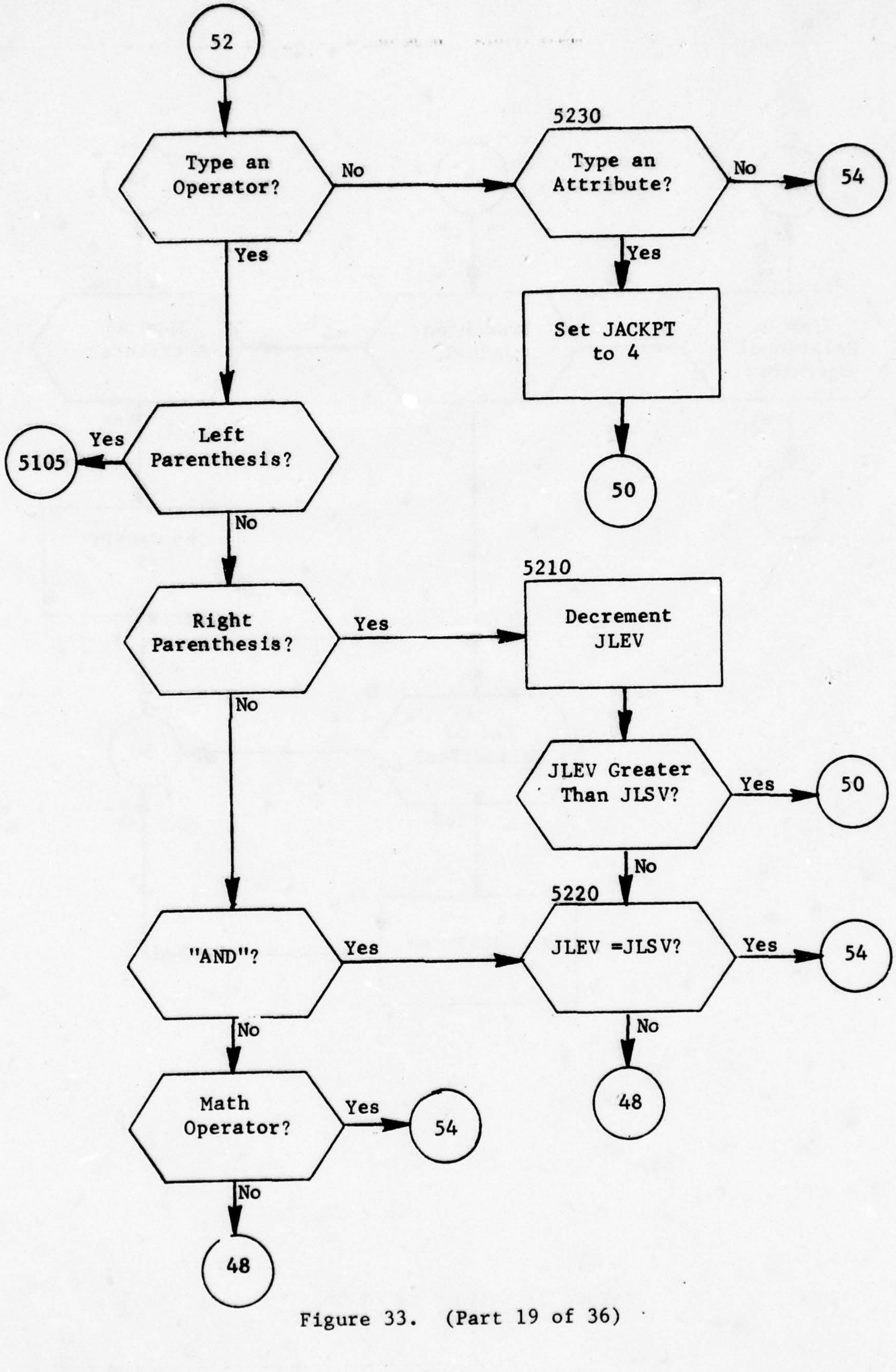



Figure 33. (Part 19 of 36)

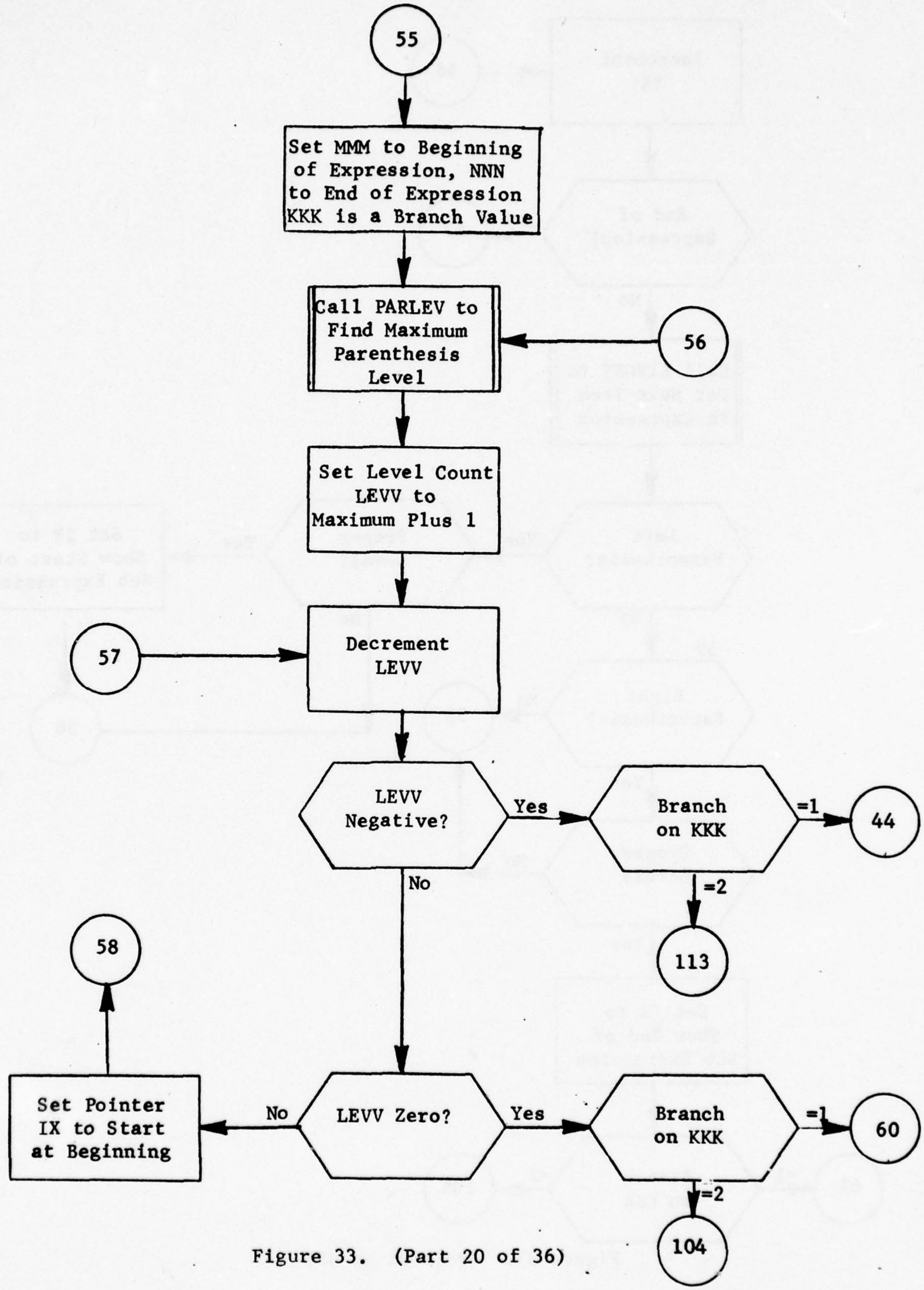



Figure 33. (Part 20 of 36)

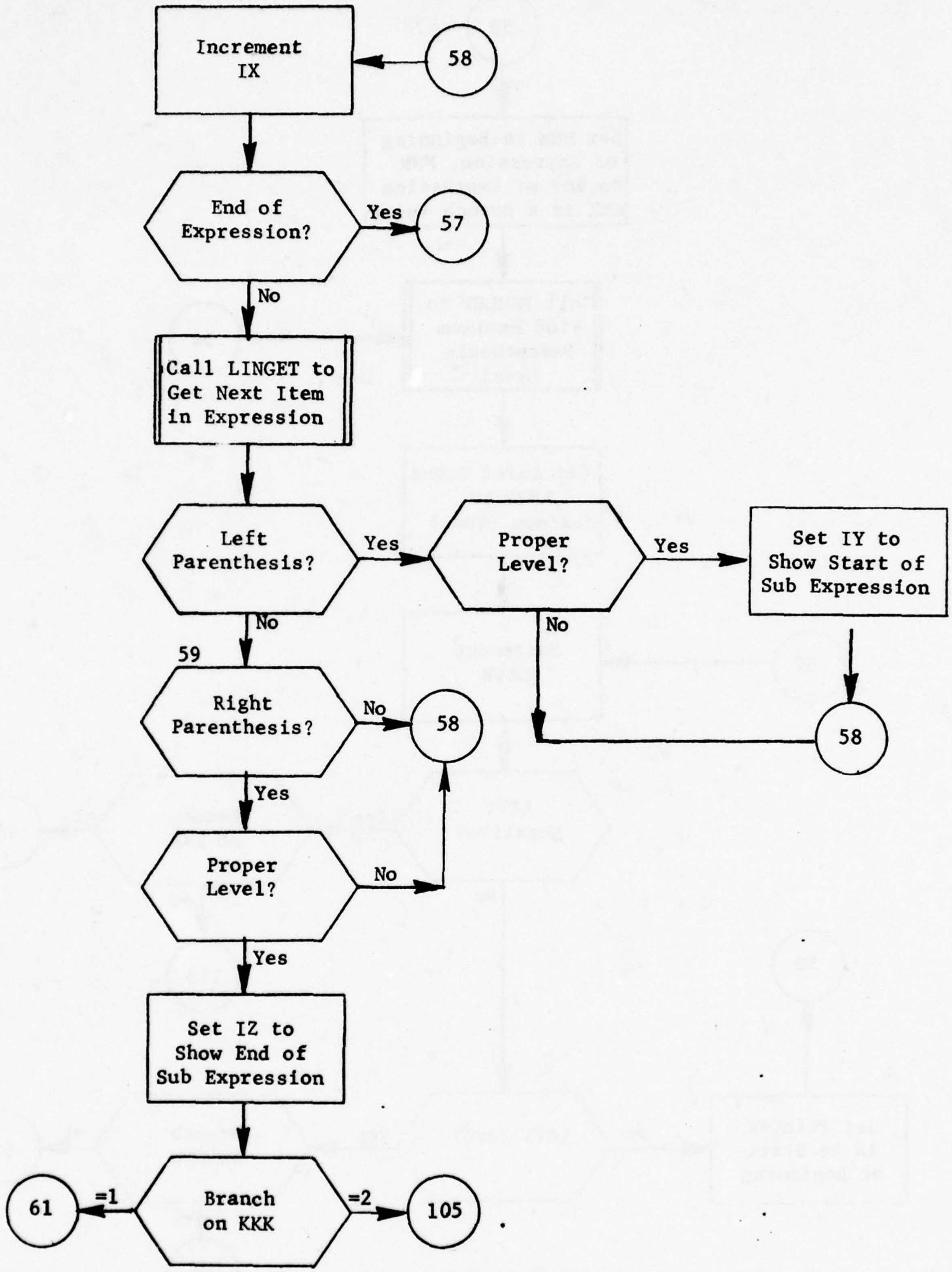



Figure 33. (Part 21 of 36)

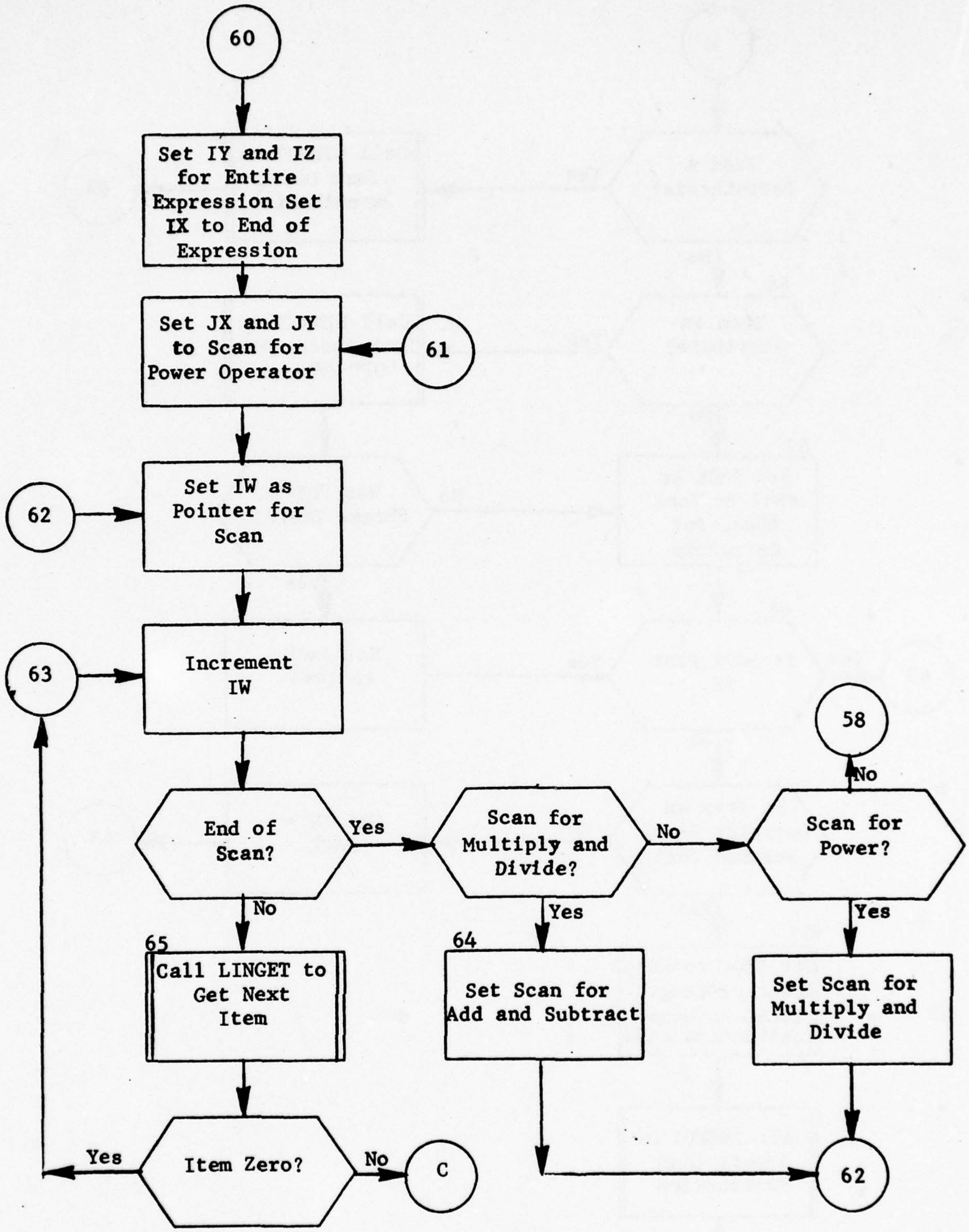



Figure 33. (Part 22 of 36)

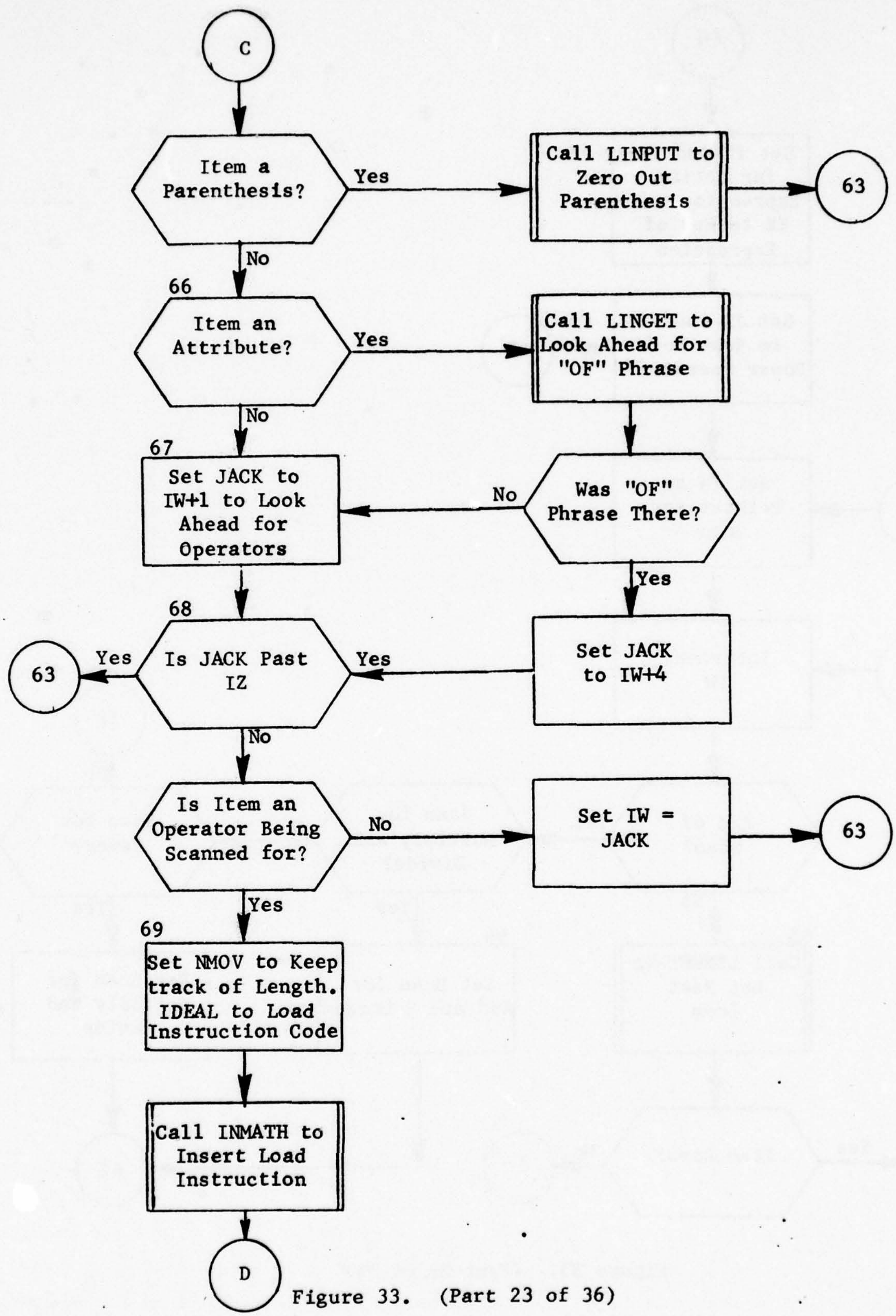



Figure 33. (Part 23 of 36)

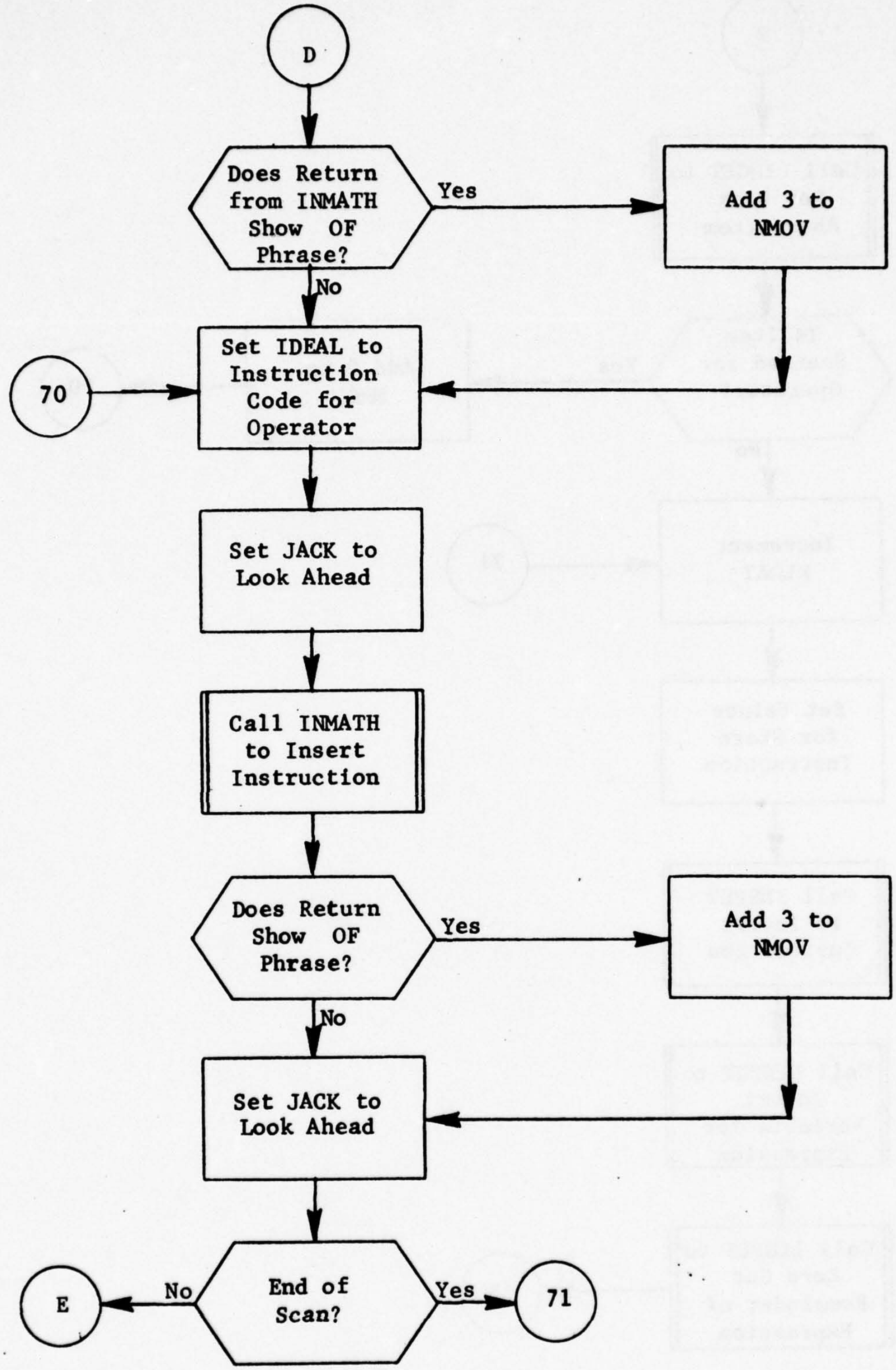



Figure 33. (Part 24 of 36)

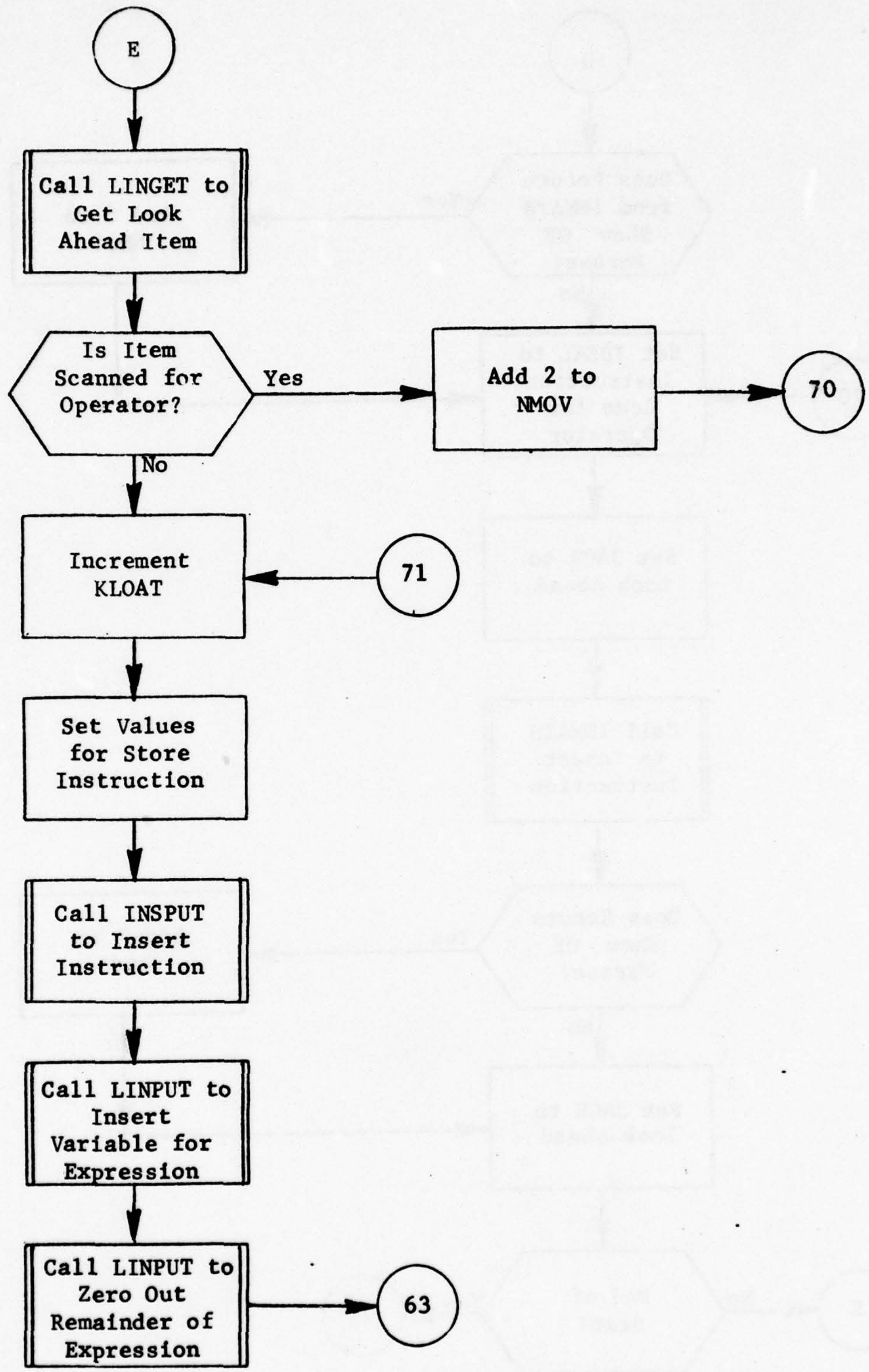



Figure 33. (Part 25 of 36)

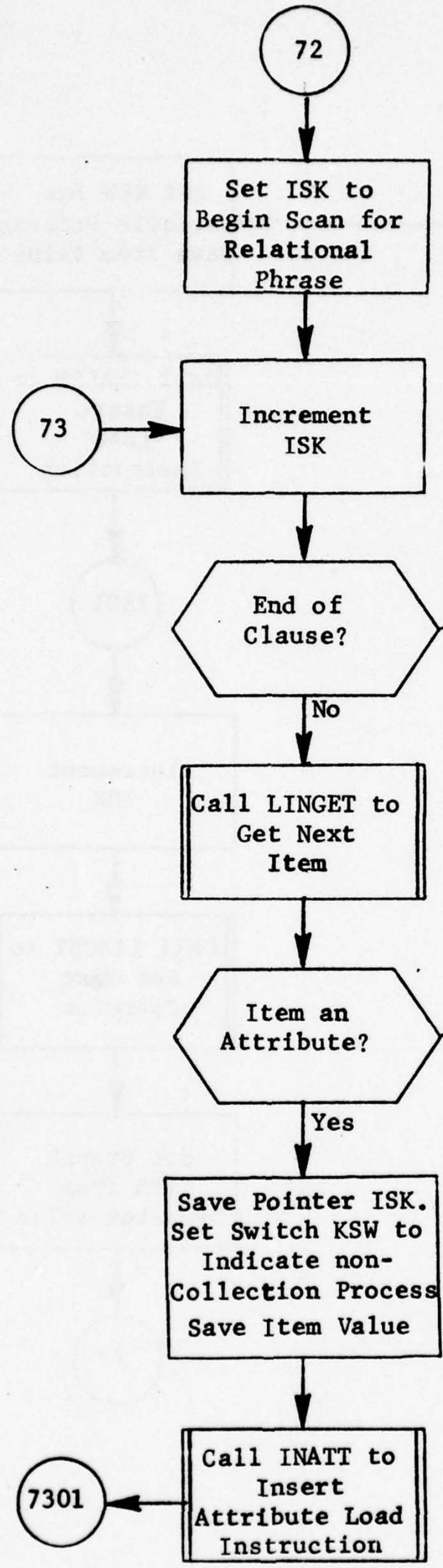



Figure 33. (Part 26 of 36)

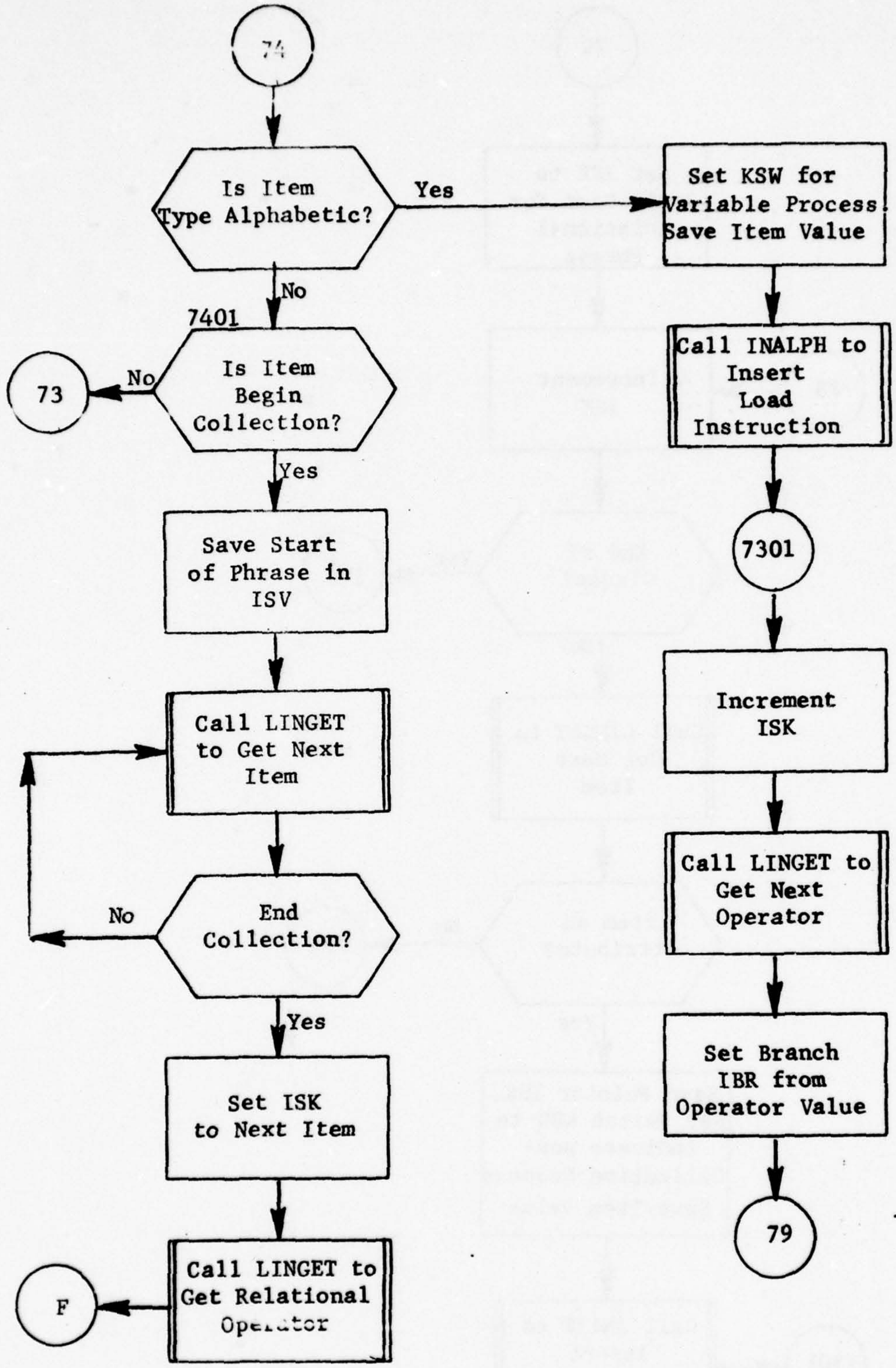



Figure 33. (Part 27 of 36)

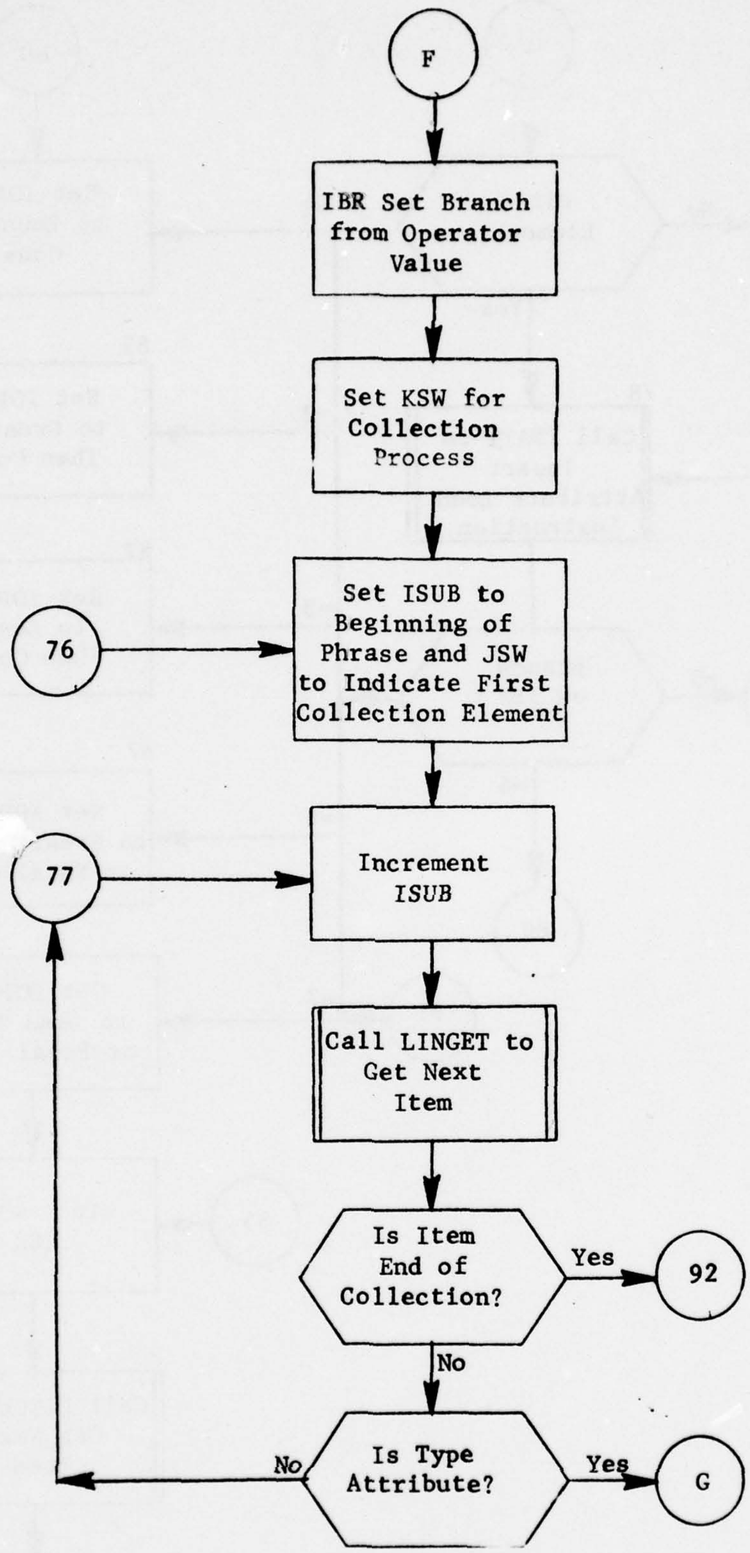



Figure 33. (Part 28 of 36)

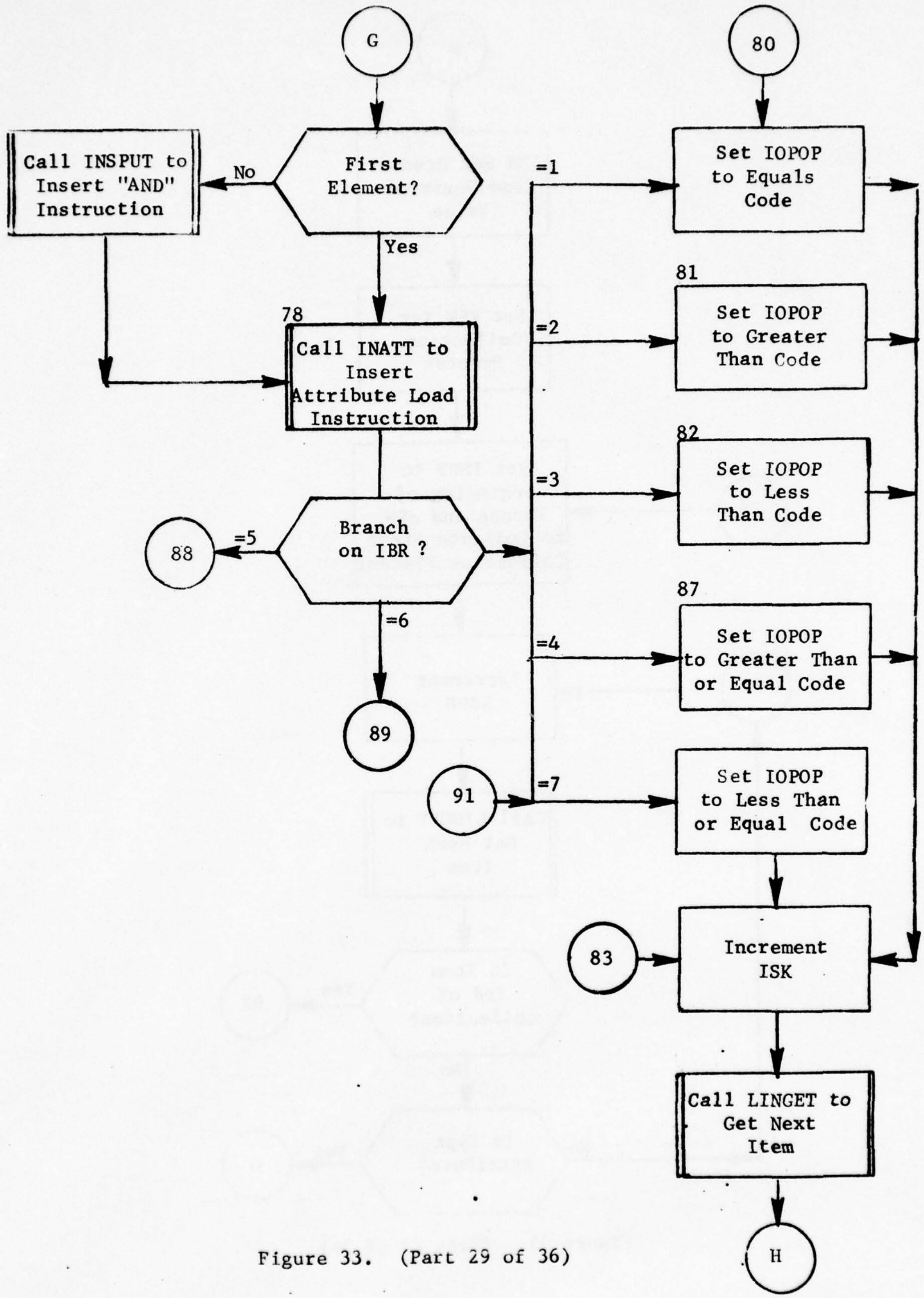



Figure 33. (Part 29 of 36)

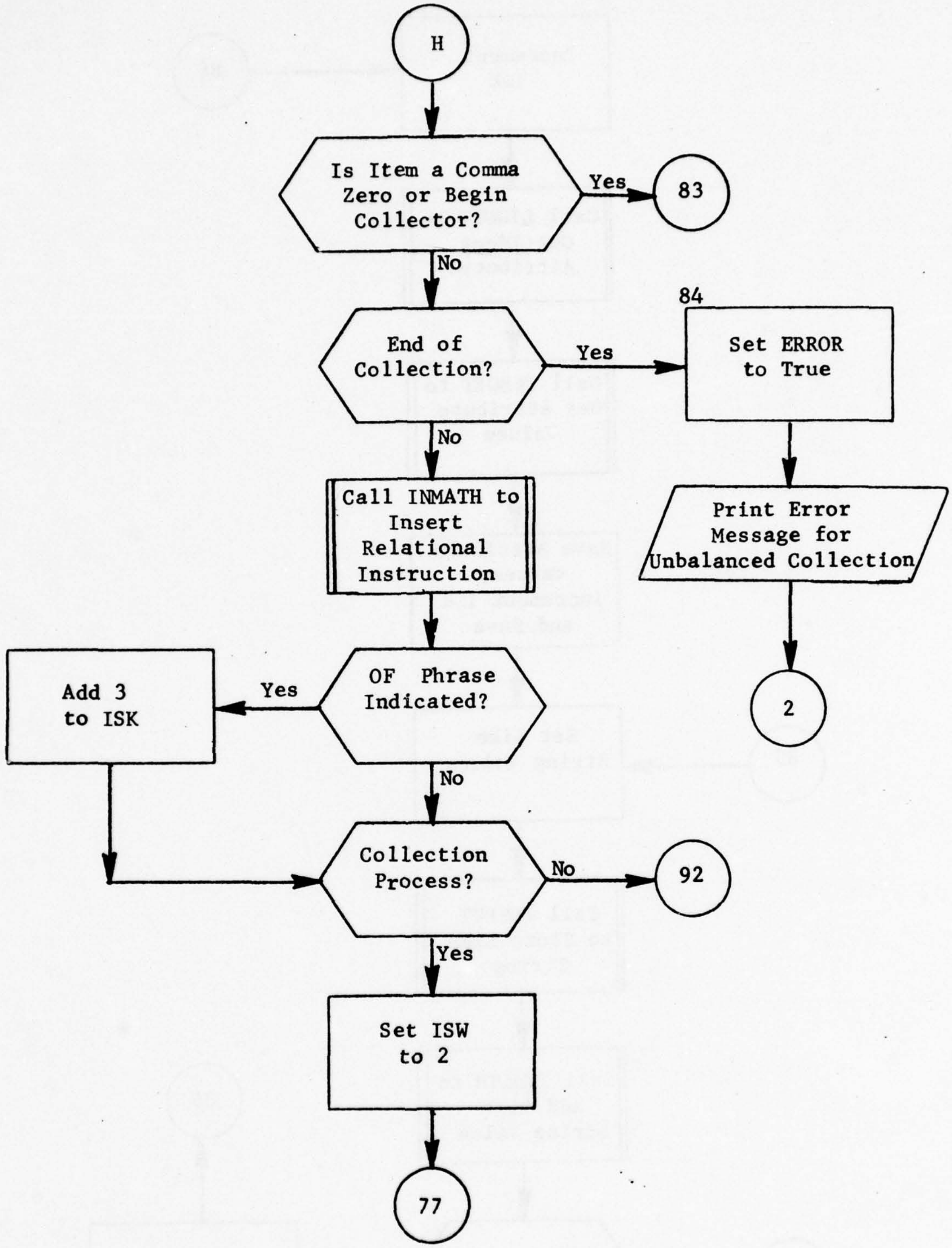



Figure 33. (Part 30 of 36)

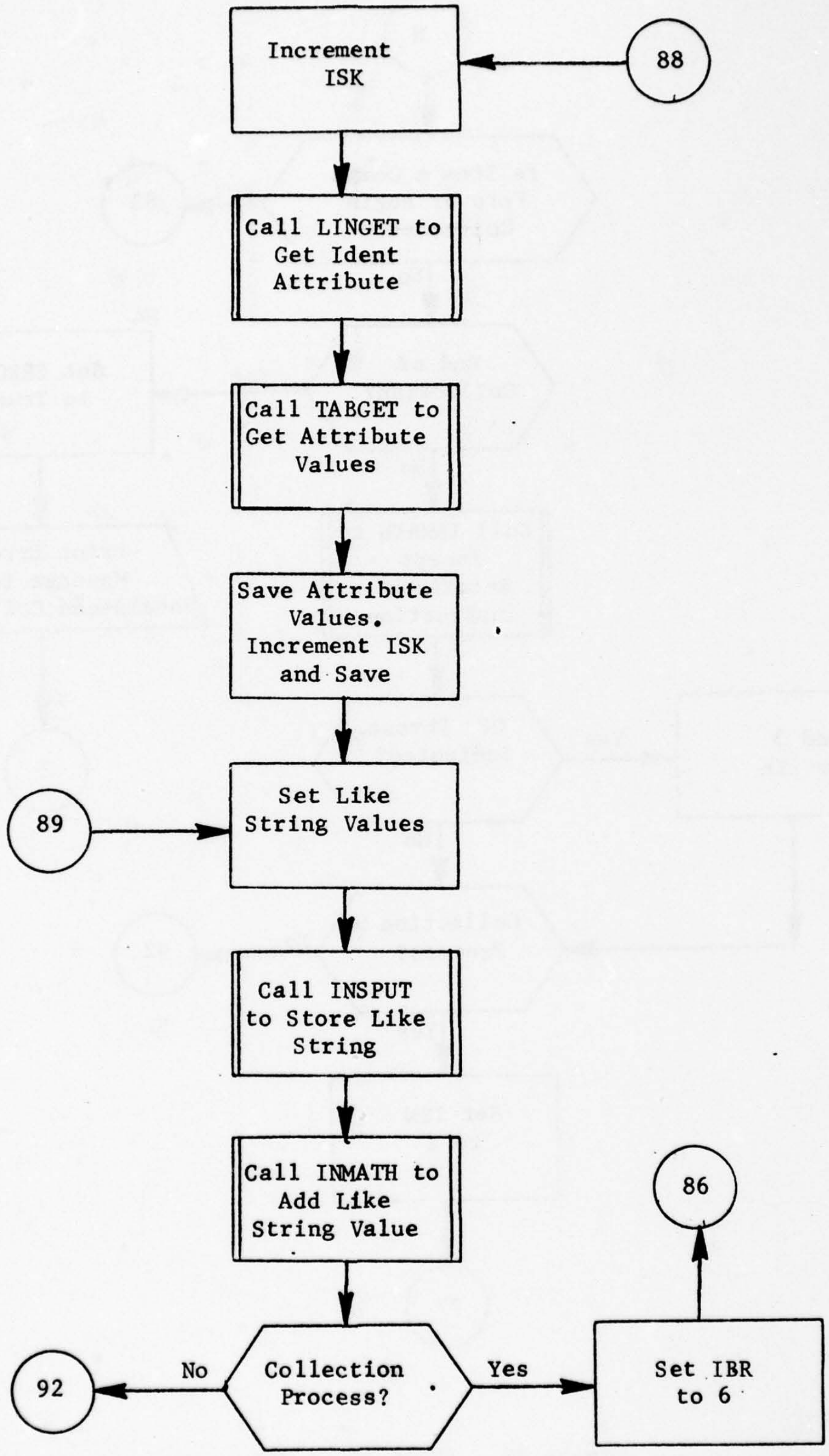



Figure 33. (Part 31 of 36)

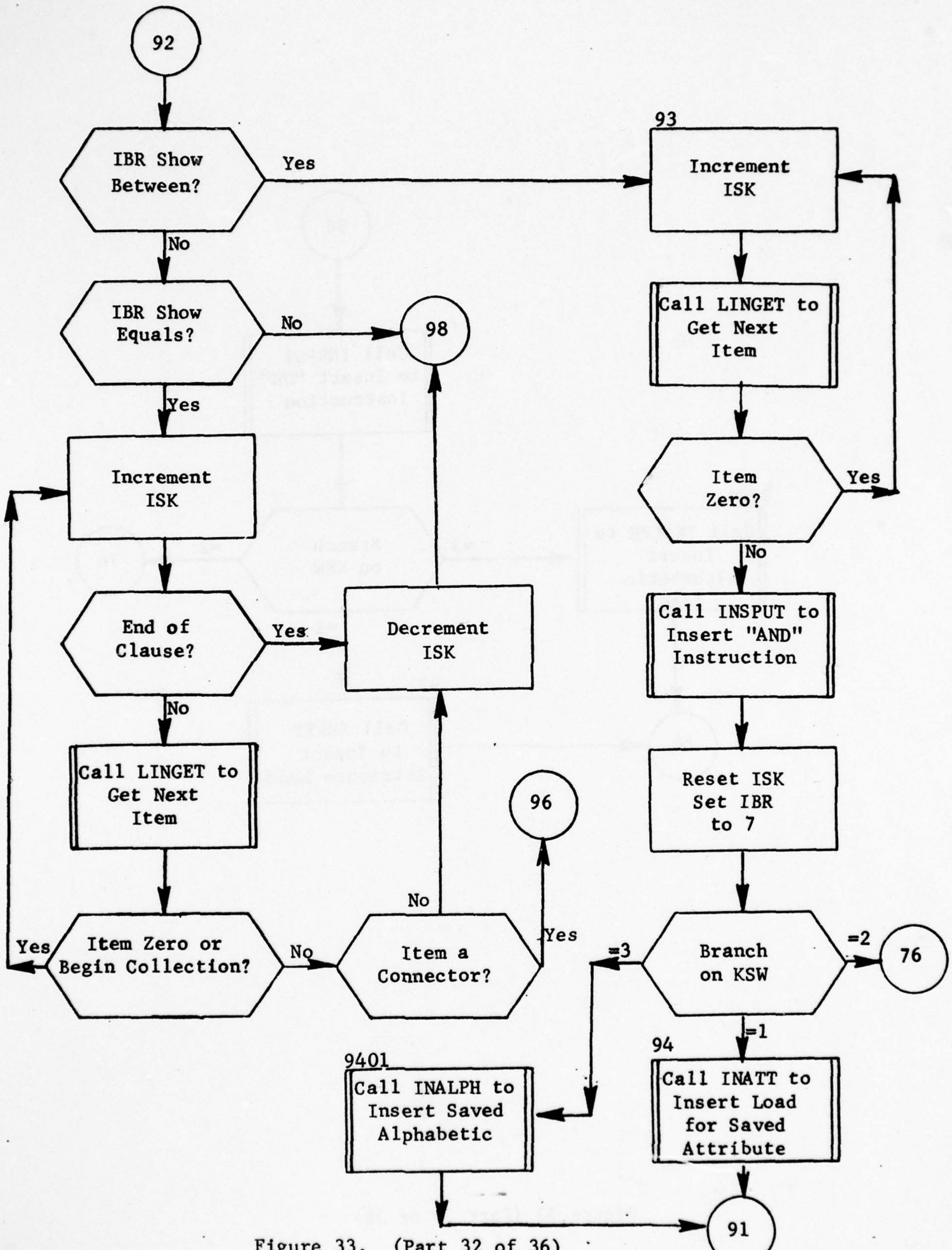



Figure 33. (Part 32 of 36)

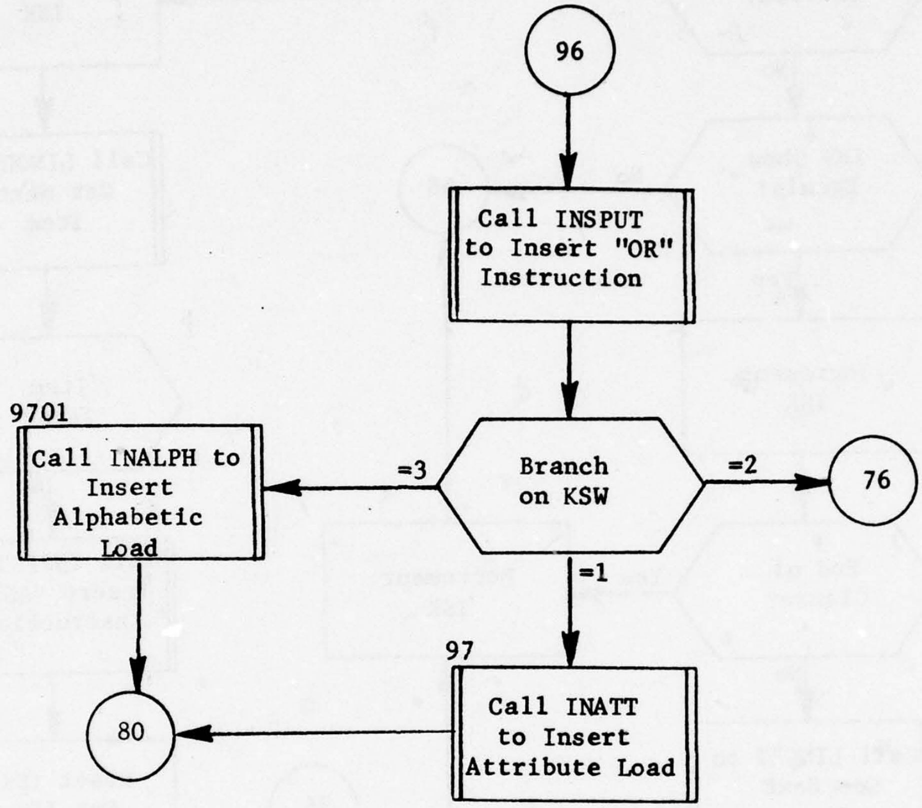



Figure 33 (Part 33 of 36)

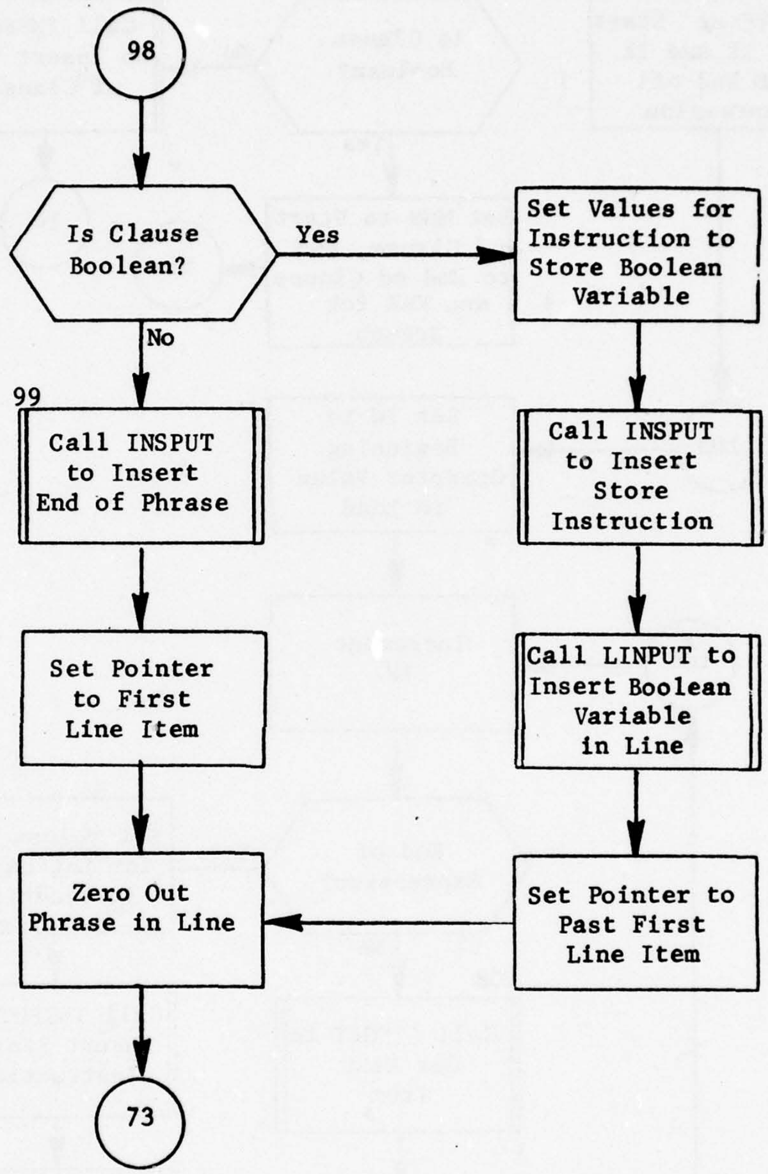



Figure 33. (Part 34 of 36)

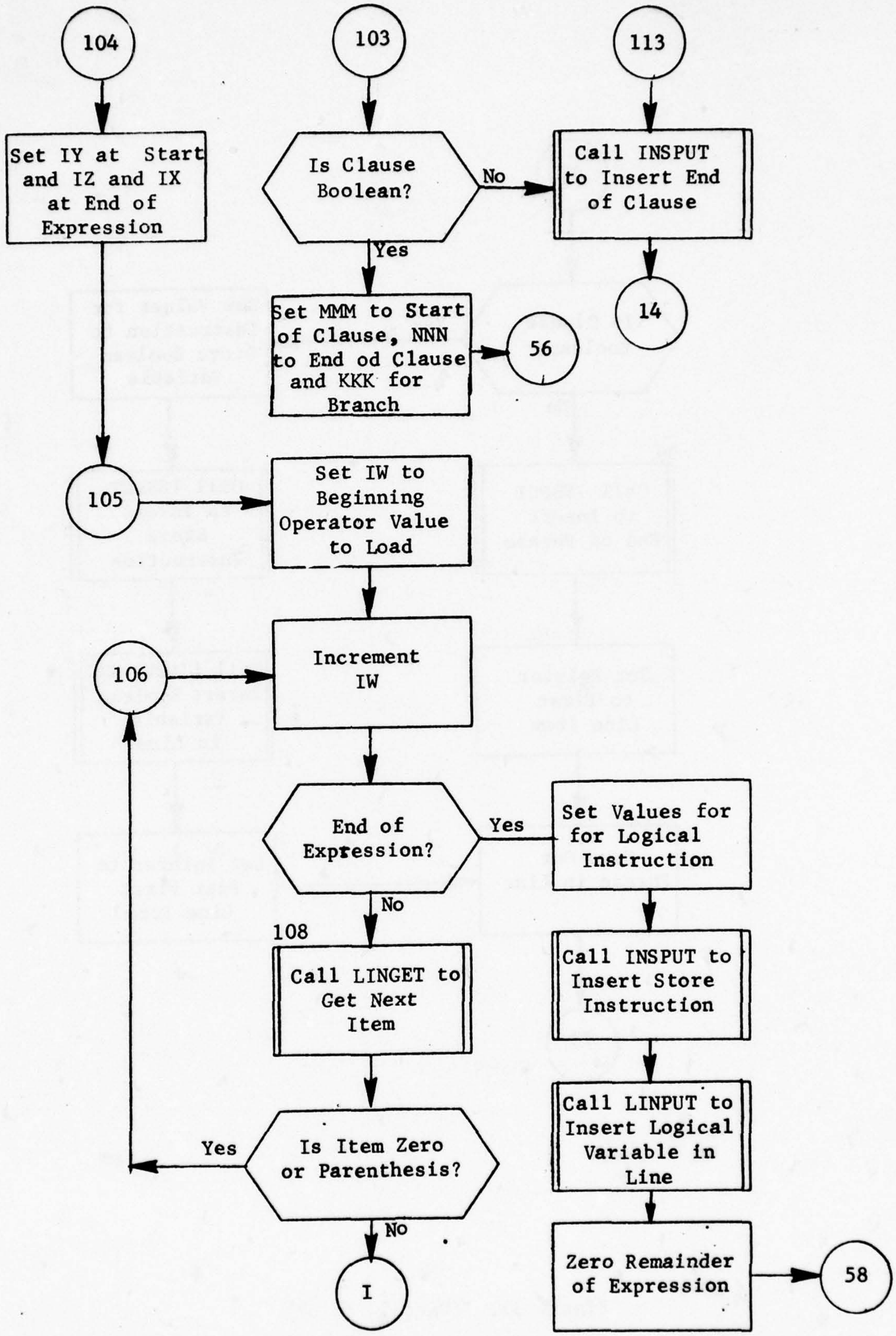



Figure 33. (Part 35 of 36)

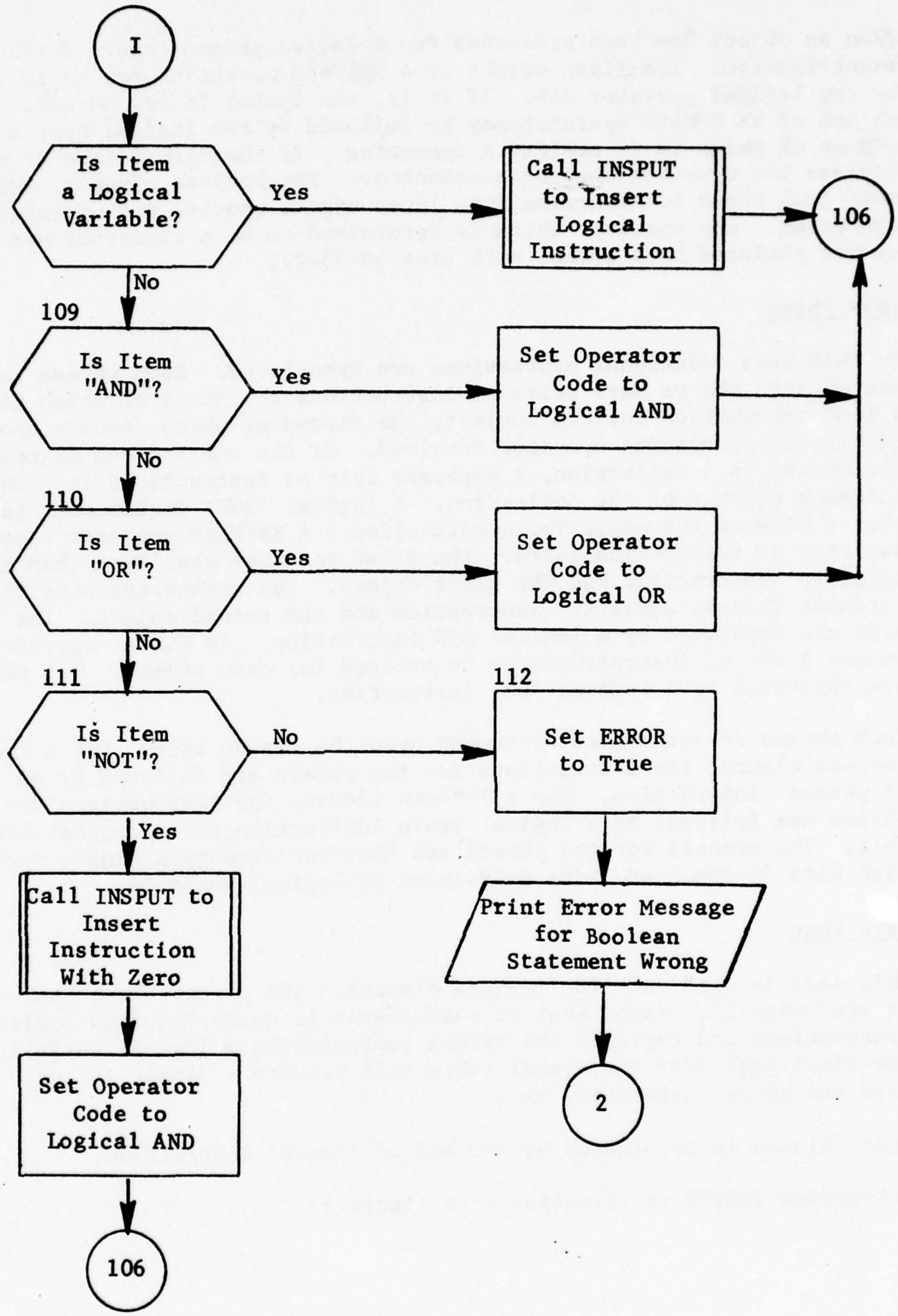



Figure 33. (Part 36 of 36)

When an object has been processed two relative phrases cause further investigation. The first object of a BETWEEN operation may be followed by the logical operator AND. If it is, the symbol is zeroed out. Any object of an EQUALS operator may be followed by the logical operator AND or OR which is in reality a connector. If the clause type is non-boolean the operator must be a connector. For boolean clauses, INPTRN must look ahead to determine from later syntax whether the operator is a connector. Any operator which is determined to be a connector has its symbol replaced by a symbol with bits 30-35=27.

## Pass Three

In this pass relational expressions are translated. Each phrase is converted into one or more pairs of instructions the first of which is a load instruction for the subject; the second of which depends upon the particular relational operator involved. If the subject and object of the phrase is a collection, a separate pair of instructions is generated for each element of the collection. A logical 'AND' instruction is placed between the pairs for a collection. A BETWEEN operator causes two sets of instruction pairs. The first set uses the 'less than or equal to' instruction and the first object. The second set uses the 'greater than or equal to' instruction and the second object. The two sets are separated by a logical AND instruction. An EQUALS operator causes a set of instructions to be created for each object. The sets are separated by a logical 'OR' instruction.

Each phrase is terminated depending upon the clause type. For a non-boolean clause, the instructions for the phrase are followed by an 'end of phrase' instruction. For a boolean clause, the instructions for the phrase are followed by a logical store instruction for a logical variable. The symbols for the phrase are then replaced by a single symbol with bits 30-35=26 and bits 0-29=index of logical variable.

## Pass Four

This pass is used only for boolean clauses. The process uses the code at statement 55. Each level of parenthesis is converted into logical instructions and replaced the symbol representing a logical variable. The final pass over the symbol table will produce a single logical load and store instruction pair.

Every clause is terminated by an 'end of clause' instruction.

Subroutine INPTRN is illustrated in figure 33.

3.10.1 Subroutine DELTAB

PURPOSE: To delete ERRFND tables

ENTRY POINTS: DELTAB

FORMAL PARAMETERS: None

COMMON BLOCKS: C40, TABLZ

SUBROUTINES CALLED: DELETE, RETRV

CALLED BY: COP, INPTRN

Method:

Each of the four types of ERRFND table is addressed separately. For each, the NUMOT (Number of old tables) Array is queried to determine the number of old tables of that type. If there are any, their reference codes are obtained from the TBRFCD array and they are retrieved and deleted.

Subroutine DELTAB is illustrated in figure 34.

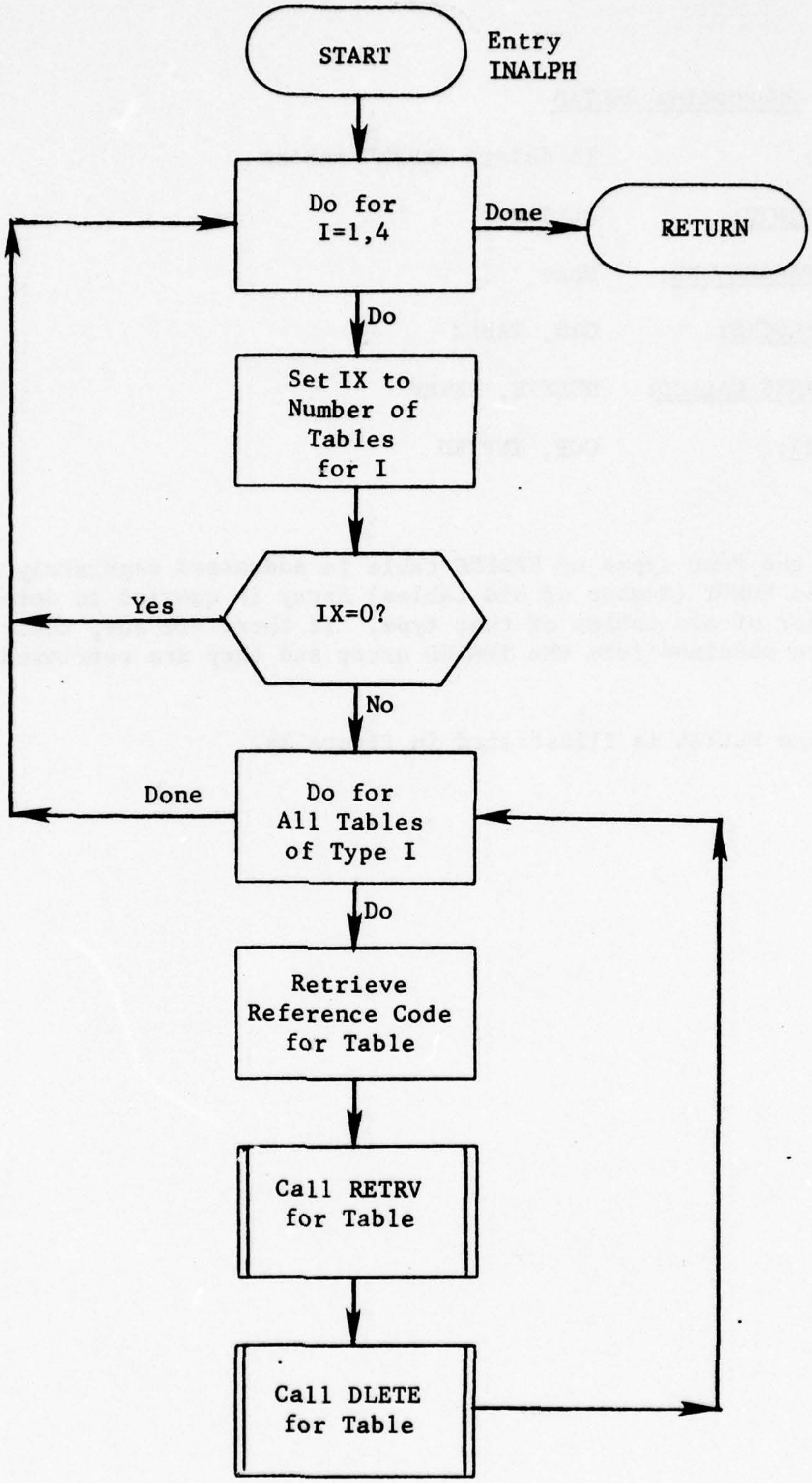



Figure 34. Subroutine DELTAB

### 3.10.2 Subroutine INMATH

| | |
|---|---|
| PURPOSE: | Create general instruction formats |
| ENTRY POINTS: | INALPH, INATT, INMATH, INUMB |
| FORMAL PARAMETERS: | INDEX: Pointer to position in input tables to start entry<br>IOP: Instruction code to be used<br>IVAL: Pointer to ERRFND table<br>ISK: Current position in input symbol table<br>ISW: Switch indicating OF phrase found<br>ITYP: Type of input symbol found |
| COMMON BLOCKS: | STRING |
| SUBROUTINES CALLED: | INSPUT, LINGET, TABGET |
| CALLED BY: | INPTRN |

Method:

The various entry points other than INMATH correspond to branches which INMATH takes once it has determined the type of the item from INGET: INUMB for numeric, INATT for attribute and INALPH for alphabetic. If this item is an internal variable the instruction code input (IOP) has 2 added to it and INSPUT is called to add the two word format.

INUMB

TABGET is called the get the numeric value and the instruction code is incremented. INSPUT is then called to add the three word format for a numeric constant in the general instruction format.

INATT

TABGET is called to get the attribute's values. The identifying number and address are unpacked and INSPUT is called to add the first four words of the general instruction format for an attribute. Now LINGET is called to look for an OF operator. If there is none, a 0 is added to the format through INSPUT. If the OF is there, ISW is set to 2 as an indication. The identifier attribute is now retrieved with LINGET and TABGET and its number and address added to the format. Then LINGET is called to get the value and its type is added to the format. Finally, TABGET is called depending on the type of value and INSPUT adds the value to the format.

INALPH

TABGET is called to get the numeric value. INSPUT is then called to add the four word format for an alphabetic constant in the general instruction format.

Subroutine INMATH is illustrated in figure 35.

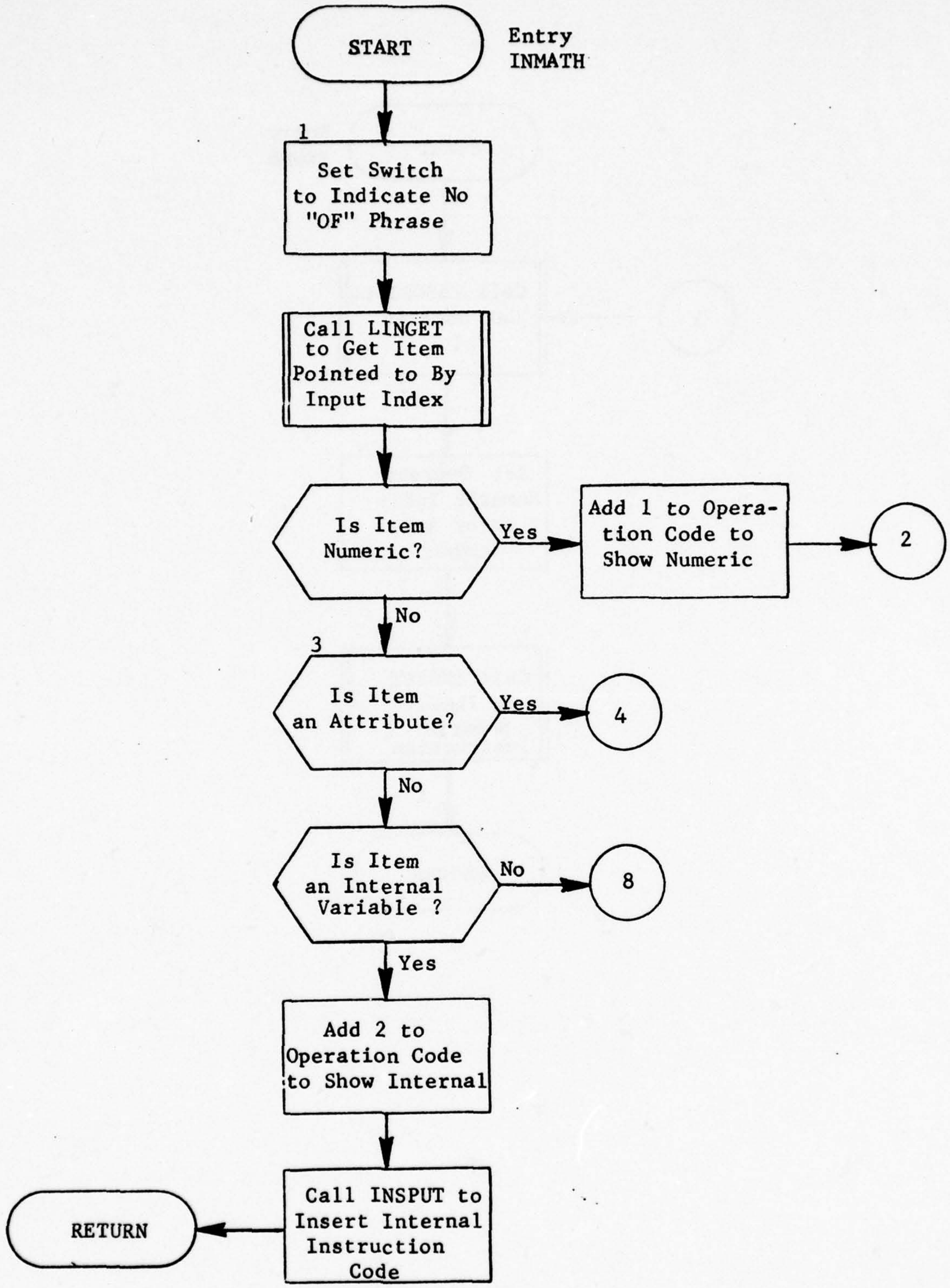



Figure 35. Subroutine INMATH: Entry INMATH (Part 1 of 5)

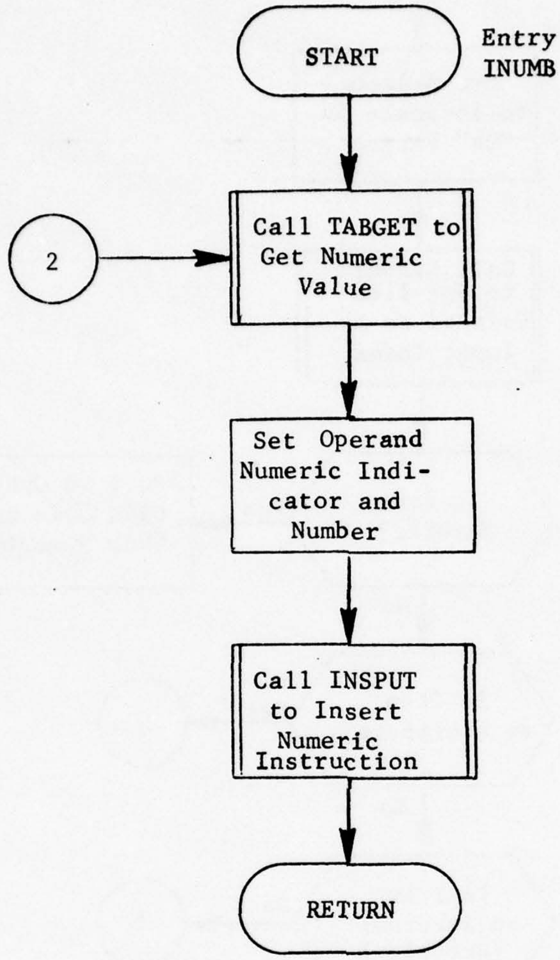



Figure 35. Entry INUMB (Part 2 of 5)

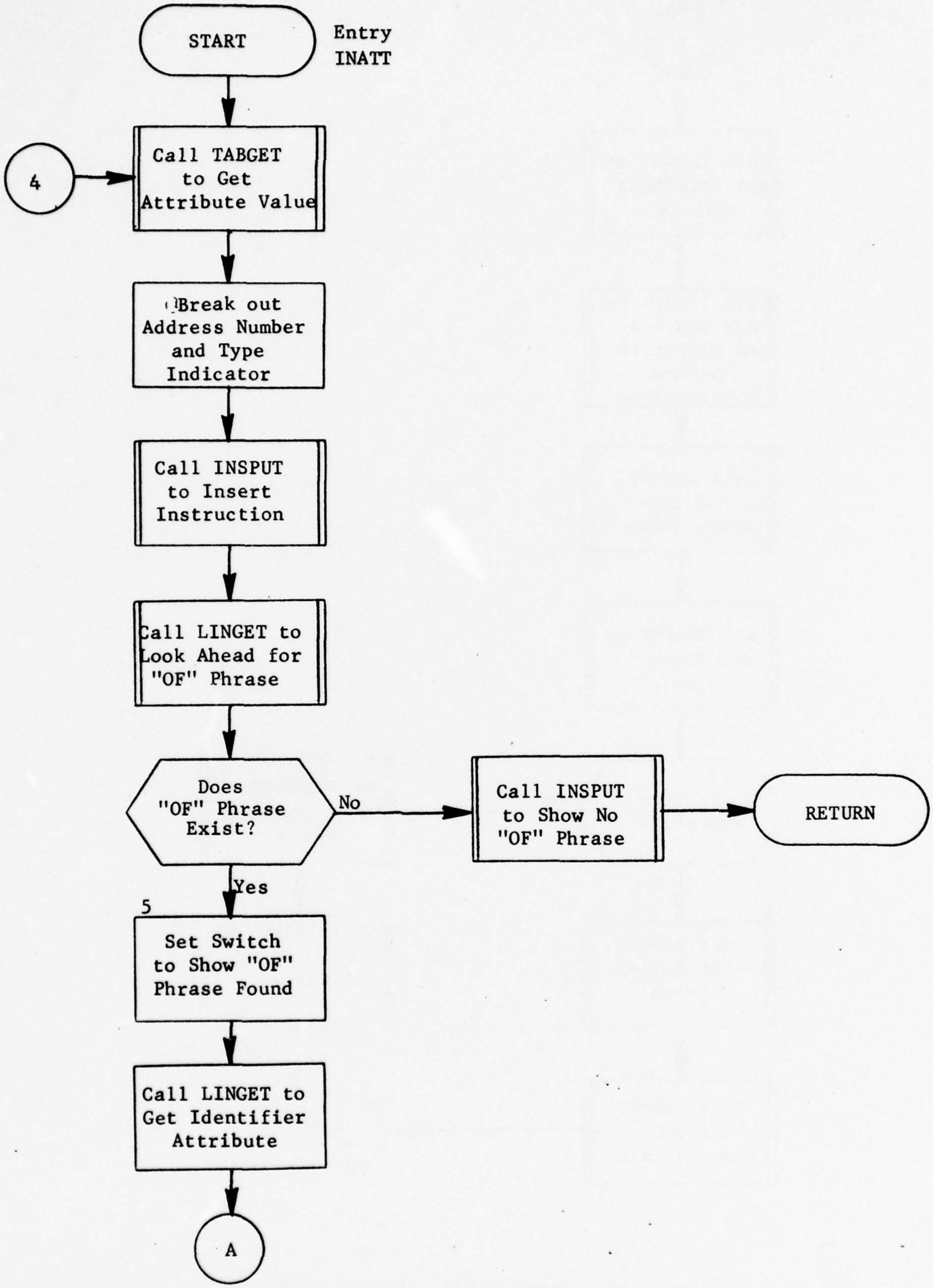



Figure 35. Entry INATT (Part 3 of 5)

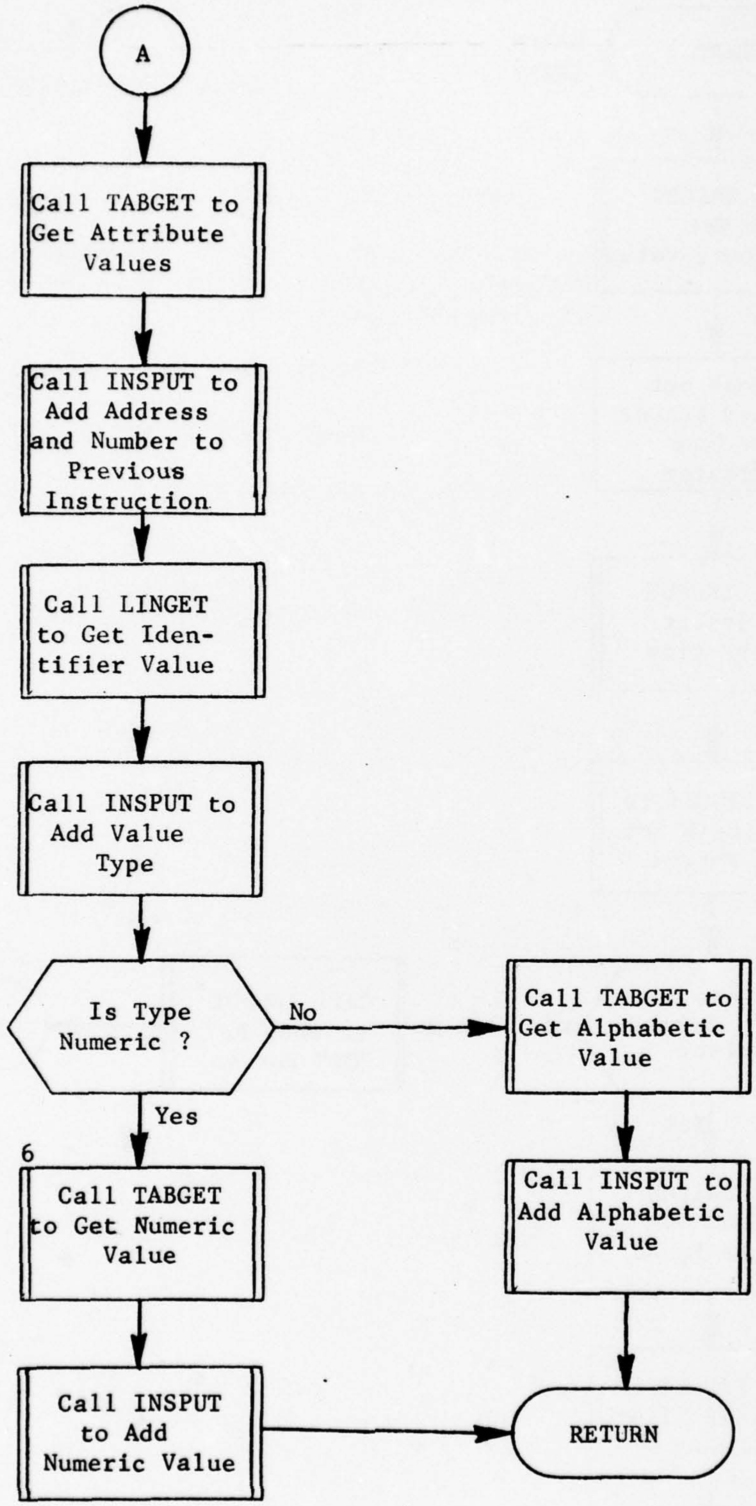



Figure 35. Entry INATT (Part 4 of 5)

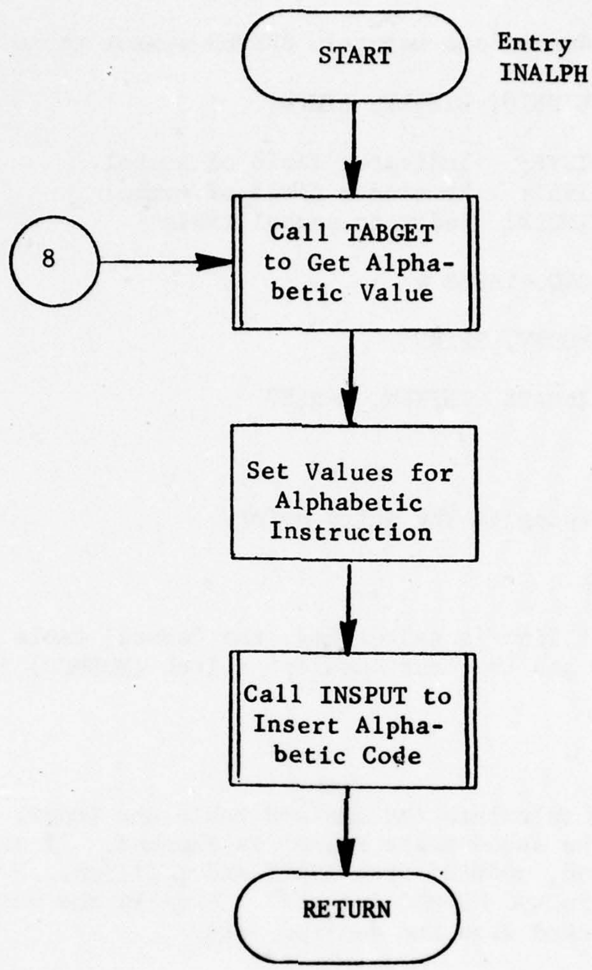



Figure 35. Entry INALPH (Part 5 of 5)

### 3.10.3 Subroutine LINEIO

| | |
|---|---|
| PURPOSE: | Access and maintain ERRFND symbol table for INPTRN |
| ENTRY POINTS: | LINEIO, LINGET, LINPUT |
| FORMAL PARAMETERS: | ITYP: Indicator field of symbol<br>IVAL: Remainder field of symbol<br>INDEX: Index to symbol table |
| COMMON BLOCKS: | C40, TABLZ |
| SUBROUTINES CALLED: | MODFY, RETRV |
| CALLED BY: | INMATH, INPTRN, PARLEV |

Method:

The method varies according to the entry point.

LINEIO

The length of the input line is calculated, the 'saved' table number (JSAVE) is set to zero and the 'must modify' switch (MUSMOD) is set to false.

LINGET

First INDEX is used to calculate the desired table and index. If the desired table is not the saved table MUSMOD is checked. If on, the saved table is retrieved, updated from ISAVE and modified. Next the desired table is read in and MUSMOD set off. Finally the values for ITYP and IVAL are unpacked from the desired word.

LINPUT

A similar process to LINGET except that when the desired word is found, ITYP and IVAL are packed into it and MUSMOD set on.

Subroutine LINEIO is illustrated in figure 36.

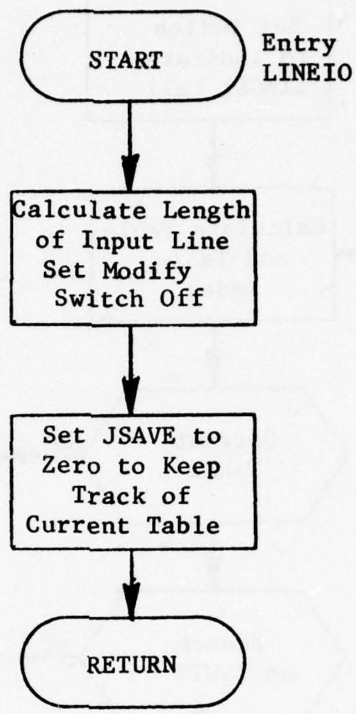



Figure 36. Subroutine LINEIO: Entry LINEIO (Part 1 of 4)

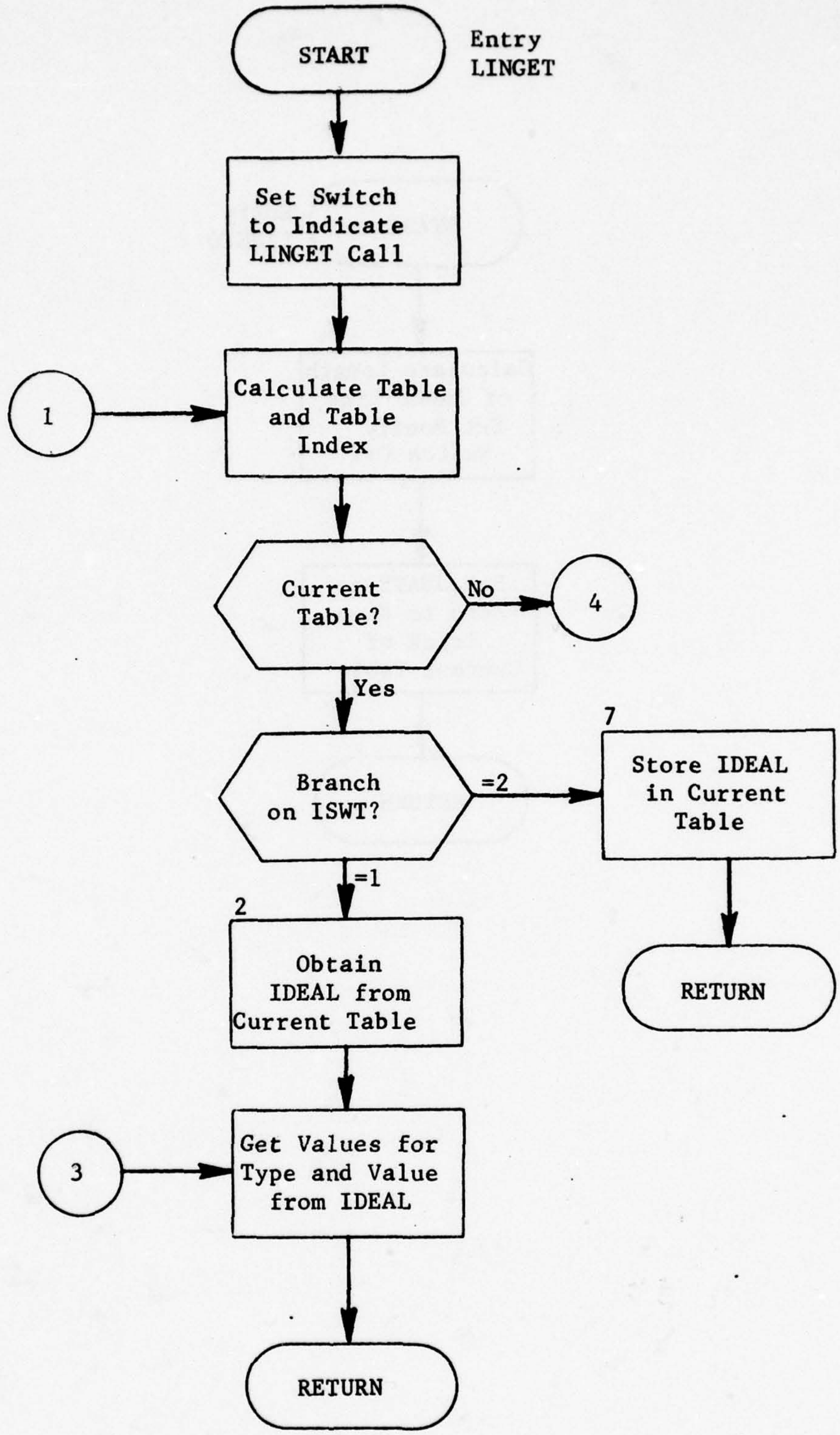



Figure 36. Entry LINGET (Part 2 of 4)

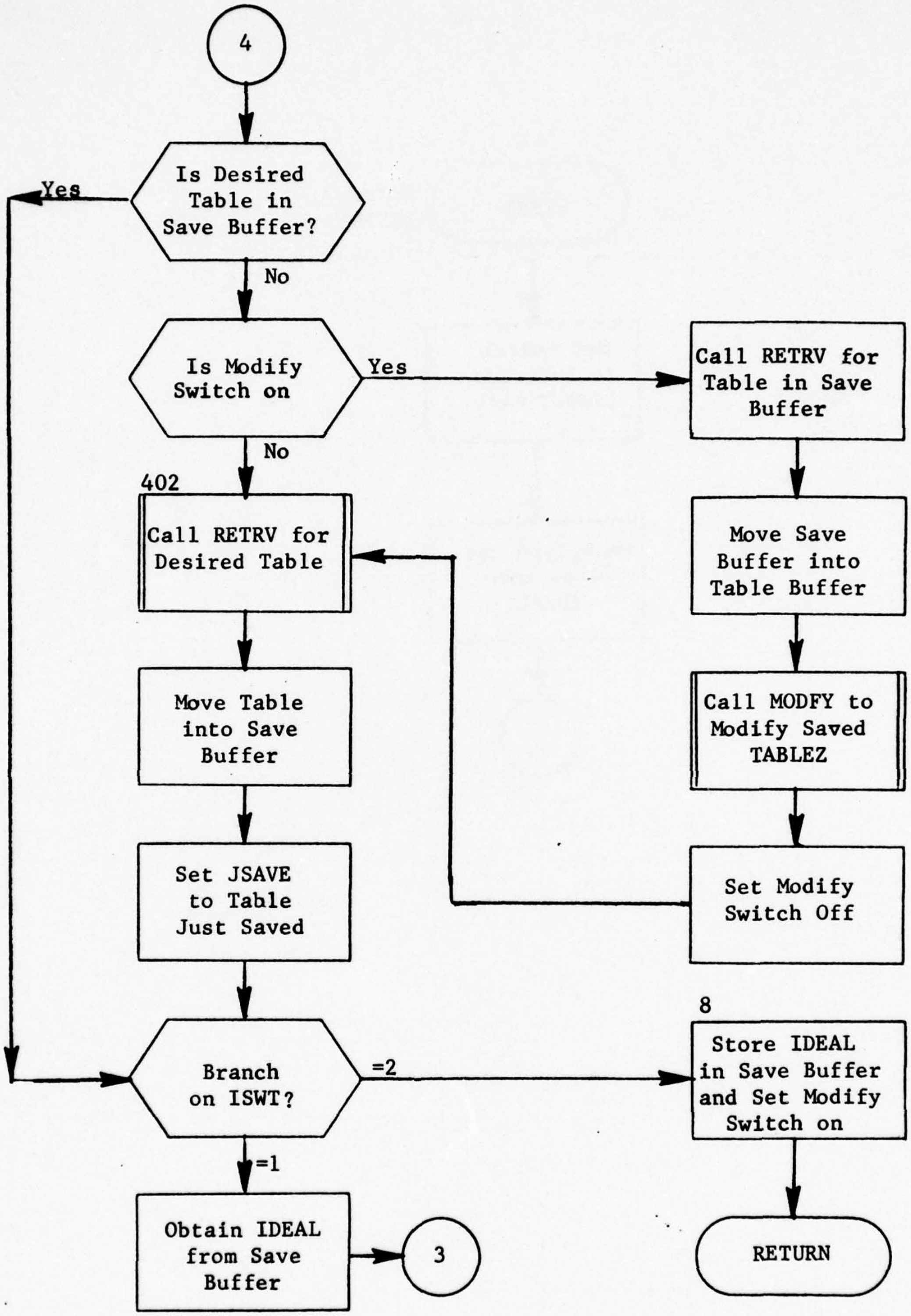



Figure 36. Entry LINGET (Part 3 of 4)

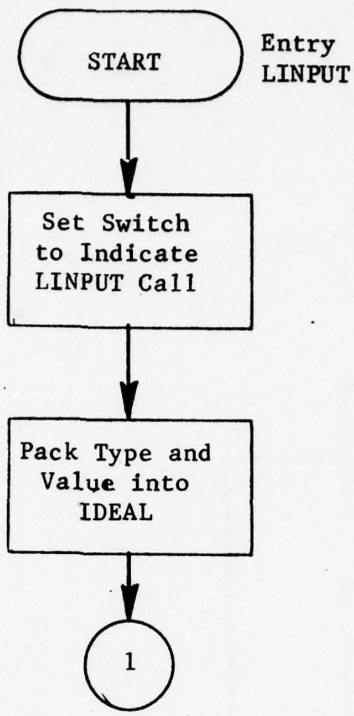



Figure 36. Entry LINPUT (Part 4 of 4)

### 3.10.4 Subroutine PARLEV

PURPOSE: To establish levels of parenthetical expressions

ENTRY POINTS: PARLEV

FORMAL PARAMETERS:

| | | |
|---|---|---|
| ISTRT: | Beginning instruction code index of expression |
| IEND: | Ending index |
| MAXLEV: | Number of levels found |

COMMON BLOCKS: None

SUBROUTINES CALLED: LINGET, LINPUT

CALLED BY: INPTRN

Method:

This subroutine scans the input symbol table from ISTRT to IEND. First two counters, one for the maximum parenthesis level (MAXLEV) the other for the current parenthesis level (LEVEL) are set to zero. As each symbol is retrieved by LINGET it is examined to see if it is a parenthesis. If it is a left parenthesis the LEVEL counter is incremented. The left parenthesis symbol is then replaced by a new symbol with bits 30-35=23 and bits 0-29=LEVEL. MAXLEV is also compared to LEVEL and set to the greater quantity.

If the symbol is a right parenthesis it is replaced by a new symbol with bits 30-35=24 and bits 0-29=LEVEL. LEVEL is then decremented.

Subroutine PARLEV is illustrated in figure 37.

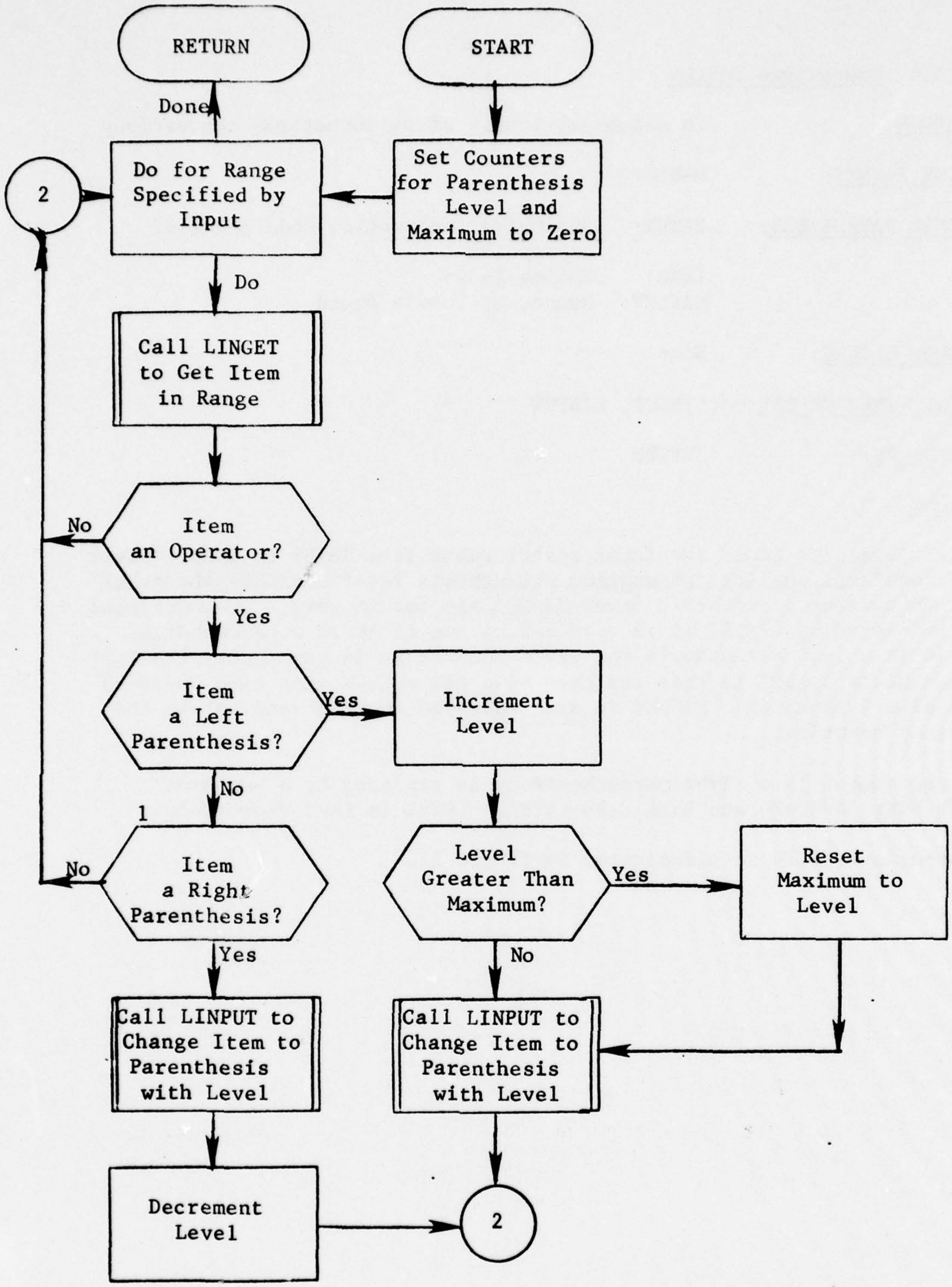



Figure 37. Subroutine PARLEV

### 3.10.5 Subroutine TABGET

| | |
|---|---|
| PURPOSE: | Obtain values from ERRFND tables |
| ENTRY POINTS: | TABGET |
| FORMAL PARAMETERS: | IXTYP: Type of table<br>INDX: Index for that table type |
| COMMON BLOCKS: | C40, STRING, TABLZ |
| SUBROUTINES CALLED: | RETRV |
| CALLED BY: | INMATH, INPTRN |

Method:

This subroutine obtains values from one of three ERRFND tables. The type of tables, indexed by IXTYP, is:

1 - Numeric constants
2 - Attributes
3 - Alphabetic constants

The process is to calculate which of the tables of the specified type is involved by dividing the input index (INDX) by the multiplier (100 for numeric or attribute, 50 for alphabetic). If this table number is greater than the number of "old tables" the value is obtained from the KTBVAL array. If the table is one of the "old tables" the old table in question is retrieved if it is not the current table and the value obtained from common block C40. The values relieved are stored in the appropriate variable of the STRING common block.

Subroutine TABGET is illustrated in figure 38.

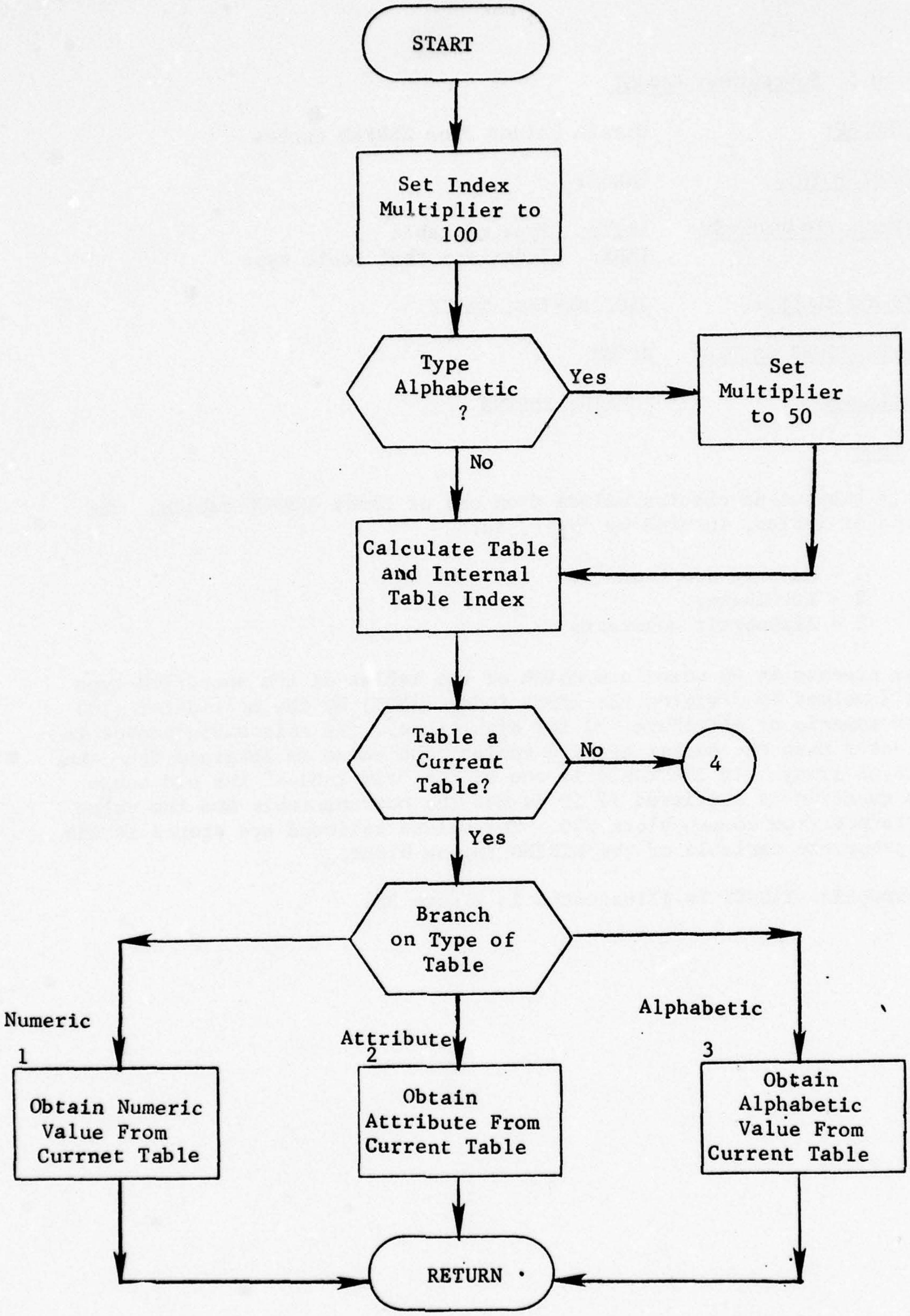



Figure 38. Subroutine TABGET (Part 1 of 2)

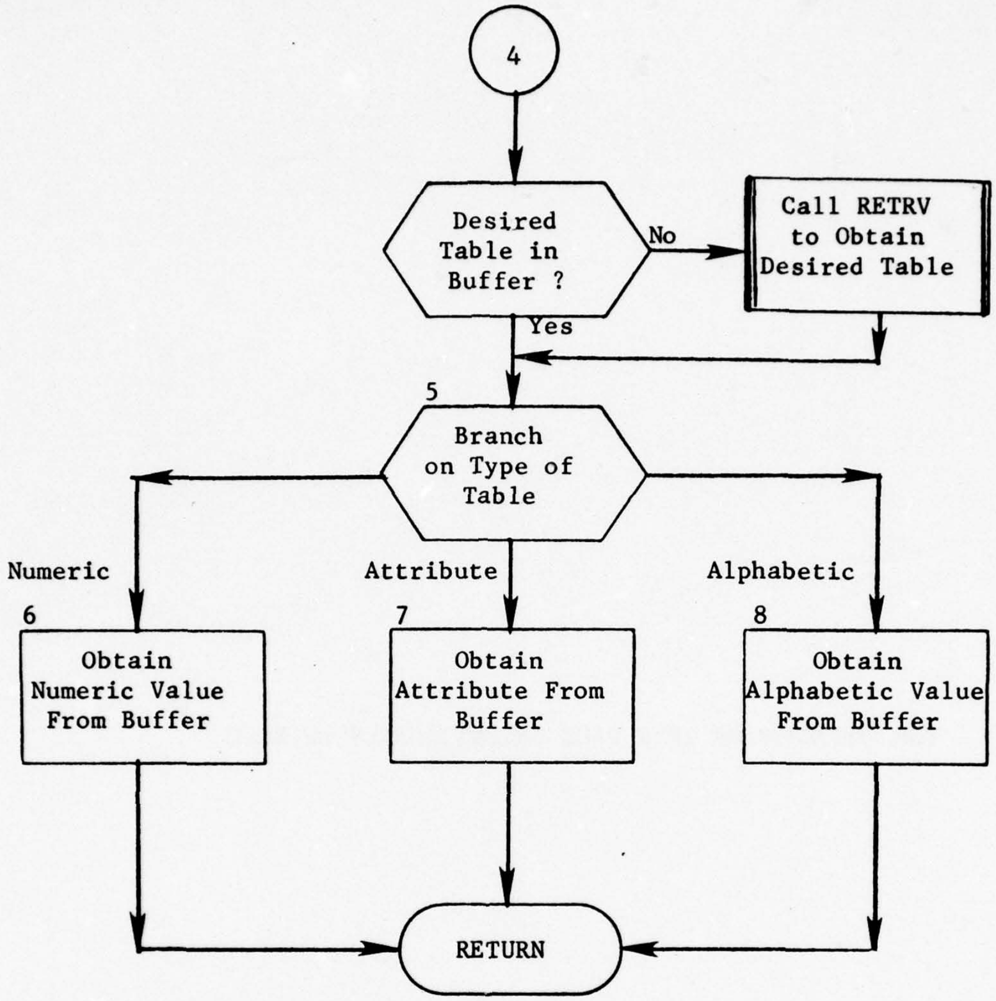



Figure 38. (Part 2 of 2)

THE CONTENTS OF THIS PAGE INTENTIONALLY DELETED

## SECTION 4. DATA MODULE

### 4.1 Purpose

The DATA module is designed to allow the user to create those portions of the data base not created by other modules, add information to existing record types which were only partially initialized by other modules, make corrections and updates to existing record types and delete record types which are no longer desired.

### 4.2 Input

The card image inputs to DATA are the user command sentences which are completely generalized to the degree possible. As with all modules, the entire integrated data base may be queried by DATA. The only data base precondition of DATA execution to the initialization of the organizational data structure. Naturally, however, logical consideration of DATA execution is a must. For instance, no record can be changed which does not already exist; the CHANGE verb will never cause new records to be created. An attempt to create any record which already exists will be ignored. Equally, an attempt to delete non-existant record types may not be honored.

### 4.3 Output

The output of DATA implies the updating (or deletion) of the contents of record types within the integrated data base. For creation, new records are added to the data base under those master and chains as directed by the user card image input. Through query of the organizational structure, DATA may readily properly place user requests. A change request alters the contents of the previously constructed records. In some cases a match key may have been changed which causes the moving of a record type from its existing chain to another chain.

### 4.4 Concept of Operation

The DATA entry module first determines which verb initiated the call. In general thereafter, the input clauses are scanned to find any included attributes which are used to determine the record types effected. Finally, whatever action the verb calls for is carried out through subroutines CREAAT, CHANGE, and DELETE.

Subsections given below describe various philosophies used in the exercising of any of the DATA verbs.

4.4.1 Retrieval Schemes. When a module wishes to retrieve a specific subset of the data base it is best to do so in an orderly fashion. That is to say, for each level of the hierarchy to be retrieved, all

associated elements of the lower levels should be retrieved before the next example of that level is retrieved. A method for doing this is the retrieval scheme.

A retrieval scheme is an array which contains a series of instruction like word groups. By following the instruction pattern contained in this array a portion of the data base will be retrieved, one unique logical set of record types at a time, until all desired logical sets have been retrieved. Each instruction consists of an introductory word which contains an identifying number, followed by one to three words which make up the remainder of the instruction. There are four instruction types: Get Header, Chain Next, Chain Master, and Return.

4.4.1.1 <u>The Get Header Instruction</u>. This instruction always occurs first in the scheme. It is the instruction that tells the executing module to look for the next data header. This instruction contains a maximum of four words depending upon the code in the second word. The first word contains a 1. The second word informs the executing module whether the attribute CLASS or SIDE (or both) are to be checked. It has the following meanings:

1 - do not check CLASS or SIDE
2 - check CLASS only. Value for CLASS appears in word three
3 - check SIDE only. Value for SIDE appears in word three
4 - check both CLASS and SIDE. Value for CLASS appears in word three; value for SIDE appears in word four

4.4.1.2 <u>The Chain Next Instruction</u>. This instruction directs the executing module to retrieve the next record on a chain and informs it as to which instruction is to be executed if the master of the chain is retrieved. The instruction always has four words. The first word contains a 2. The second word contains the name of the chain. The third word contains the record type number of the chain's master. The fourth word contains a pointer (index number) to the instruction to be executed if the master of the chain is retrieved. In most cases, the pointer will be to the previous Chain Next Instruction or the Get Header Instruction.

4.4.1.3 <u>The Chain Master Instruction</u>. This instruction directs the executing module to retrieve the master record of a chain. The instruction always has two words. The first word contains a 3. The second word contains the name of the chain.

4.4.1.4 <u>The Return Instruction</u>. This instruction always appears last in a scheme. It informs the executing module that a logical set of records has been retrieved. It also informs the module of the instruction to execute next to retrieve the next logical set of records. The instruction contains two words. The first word contains a 4. The second word contains a pointer to a previous instruction.

4.4.1.5 Retrieval Supporting Subroutines. Retrieval schemes are built by utility subroutines SETSCH from a continuous set of record type numbers. By continuous is meant that every record type in the set is either the master or detail of a chain or series of chains by which it is linked to every other record type in the set. The utility subroutine LINKUP is designed to assist in this process. To build a retrieval scheme LINKUP and SETSCH also require that one of the record types in the set be identified as the 'primary header'.

4.4.2 Primary Header. The QUICK integrated data base is designed with the idea that all data will be retrieved through the use of data entry point record types known as headers (see section 2). The set of record types from which a retrieval scheme may be built, might conceivably contain any number of such headers. Therefore, one of the headers must be designated as the data entry point and is referred to as the 'primary header.' The DATA module uses several methods to determine the primary header. One is to use the desired value for the attribute CLASS. Since this attribute uniquely defines a header, DATA assumes this is the primary header. Another method is to choose some lower level record type in the hierarchy. This record may be determined by record type number since as the record type numbers increase the hierarchial level tends to decrease or, in the case of CREAAT, by the record type with the greatest number of involved attributes. In either case utility subroutine PRIMHD is called. This subroutine traces chains of which the record type is a detail up to a header. This header is then used as the primary header.

4.4.3 Determining a Record Type Set from a List of Attributes. The set of record type numbers is built by using the list of attributes which have been included in the input clauses. Attributes are of three types insofar as the record types that include then are concerned and are:

Single - This type of attribute is included on only one record type.

Control - This type of attribute is included on several record types but one of the record types is uniquely identified by the attribute. The record type so identified is referred to as "controlled."

Multiple - This type of attribute is included on several record types but is not a unique identifier for any of them.

The general method of determining a record set from the list of attributes is to make two record type lists. The first or main list includes all those record types which contain single attributes in the attribute list plus the controlled record types of any control attributes in the attribute list. The second or multiple list includes all record types which include any of the multiple attributes in the attribute list. The primary header is now found either from a value for CLASS or from

one of the record types in the main list. The primary header is then used in a 'chain-down' process. This type of process consists of checking every record type which is a detail of chains of which the primary header is the master. Then using these detail records as masters of chains and so forth until the lowest levels of the hierarchy headed by the primary headers have been reached. Throughout the process, the multiple list is examined and anytime a record type in the hierarchy appears in the multiple list it is added to the main list. The main list is then used as the set of record types from which a retrieval scheme is built.

4.4.4 Data Queus. In the process of the CREAAT or CHANGE subroutines often the subroutine must build one or more data queues. This occurs when an attribute or collection of attributes are given more than one value or collection of values. The queue is an ordered list of the values given to the attribute or collection and is maintained by the subroutine VALPUT. Each value assigned to a particular attribute can be identified by its position in the queue. Each value assigned to the member of a collection can be identified by its position within the collection plus its position in the queue minus one times the length of the collection.

The CREAAT subroutine uses the set of queues it has created to determine how many times it must go through the process of creating record types. For each combination of values in queues, the process must be followed. For example, if there is one queue for a single attribute with three values and another queue for a collection of four attributes with five sets of values (a total of 20 values), the record creation process would be performed 15 times. CREAAT uses a set of queue counters which are all originally set to 1 indicating the first value or set of values in each queue. After every execution of the record creation process, the last queue counter is incremented unless it is at maximum. In this case it is reset to one and the next to last counter incremented. If the next to last is at maximum it is reset to one and the prior counter incremented and so on. In this way all combinations are used in the record creation process.

The CHANGE subroutine expects that if a data queue is formed from the SETTING clause one of the same length will be formed from the WHERE clause. Then when a record is retrieved which satisfies the WHERE clause the data queue for the WHERE clause is searched until the value (or collection of values) which caused the match is found. The value (or collection of values) which occupies the same ordinal position in the data queue for the SETTING clause is used in the record change process.

## 4.5 Identification of Subroutine Functions

4.5.1 CREAAT. This subroutine (figure 39) performs the functions of creating new records. After checking for the existance of SAME and SUPRESSING adverbs, CREAAT scans the SETTING clause (any number of SETTING clauses may be input, each is processed similarly). First the attributes are collected, calculations flaged and OF phrases resolved. Next, attribute values are saved and data queues built for attributes provided with multiple values. Next a retrieval scheme is built by identifying the record types defined by the attributes. Input values are checked against QUICKs directory. Attributes on records to be created but not given values are assigned values either according to the defaults in the directory or from the record type(s) identified by the SAME clause. Finally, records are created of every type identified except in cases where the record to be created would duplicate an existing record.

4.5.2 CHANGE. This subroutine (figure 40) performs the function of changing existing records. First the SETTING clause is scanned and attributes and values are collected. If a data queue exists it is set up. Next the WHERE clause is scanned and its attributes are collected. The two clause's lists are combined with any WHERE clause queue paired with its SETTING clause counterpart. The list of attributes is used to build a retrieval scheme. The retrieval scheme is now executed and any record type combination retrieved which satisfies the WHERE clause is modified according to the SETTING clause.

4.5.3 DELETE. This subroutine performs the function of deleting unwanted records. First the WHERE clause is scanned for attributes. Next a retrieval scheme is built using the attributes found. From this scheme, the record lowest in the hierarchy is determined. Finally, the scheme is executed and for every record type combination retrieved which satisfies the WHERE clause, the lowest record type in the hierarchy is deleted.

## 4.6 Common Blocks

The common blocks internal to the DATA module appear in table 15.

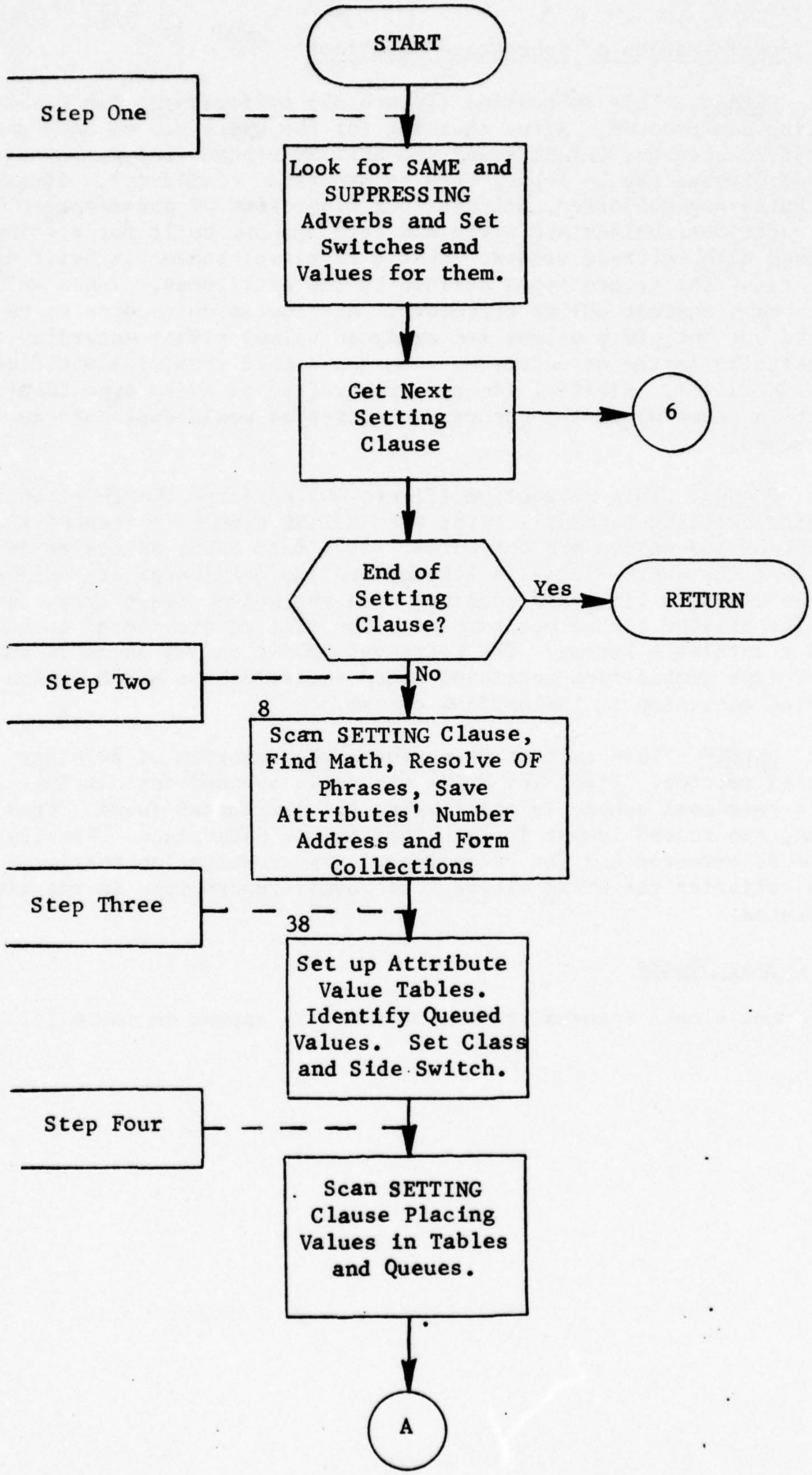



Figure 39. Subroutine CREAAT: Macro Flow (Part 1 of 3)

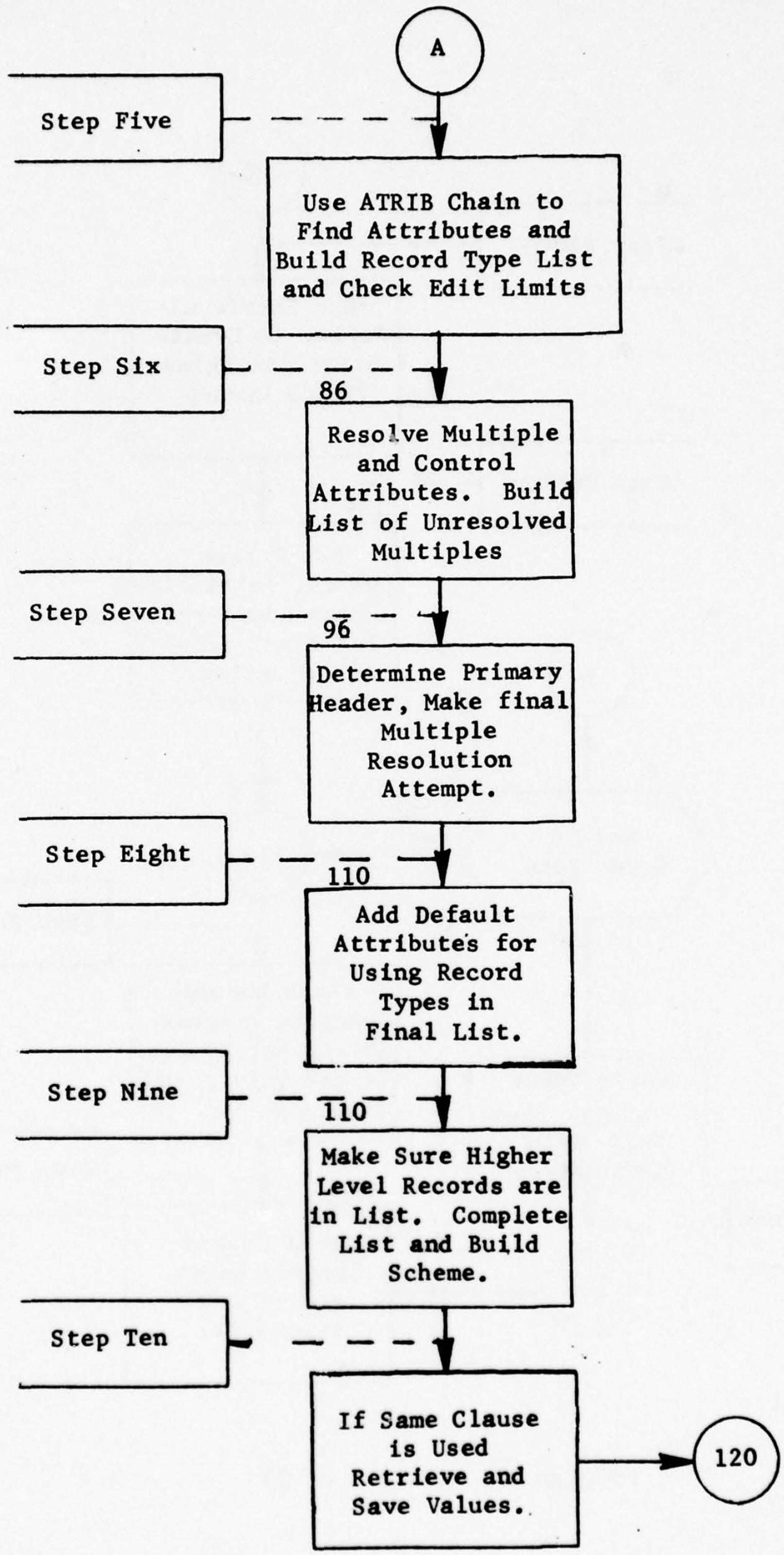



Figure 39. (Part 2 of 3)

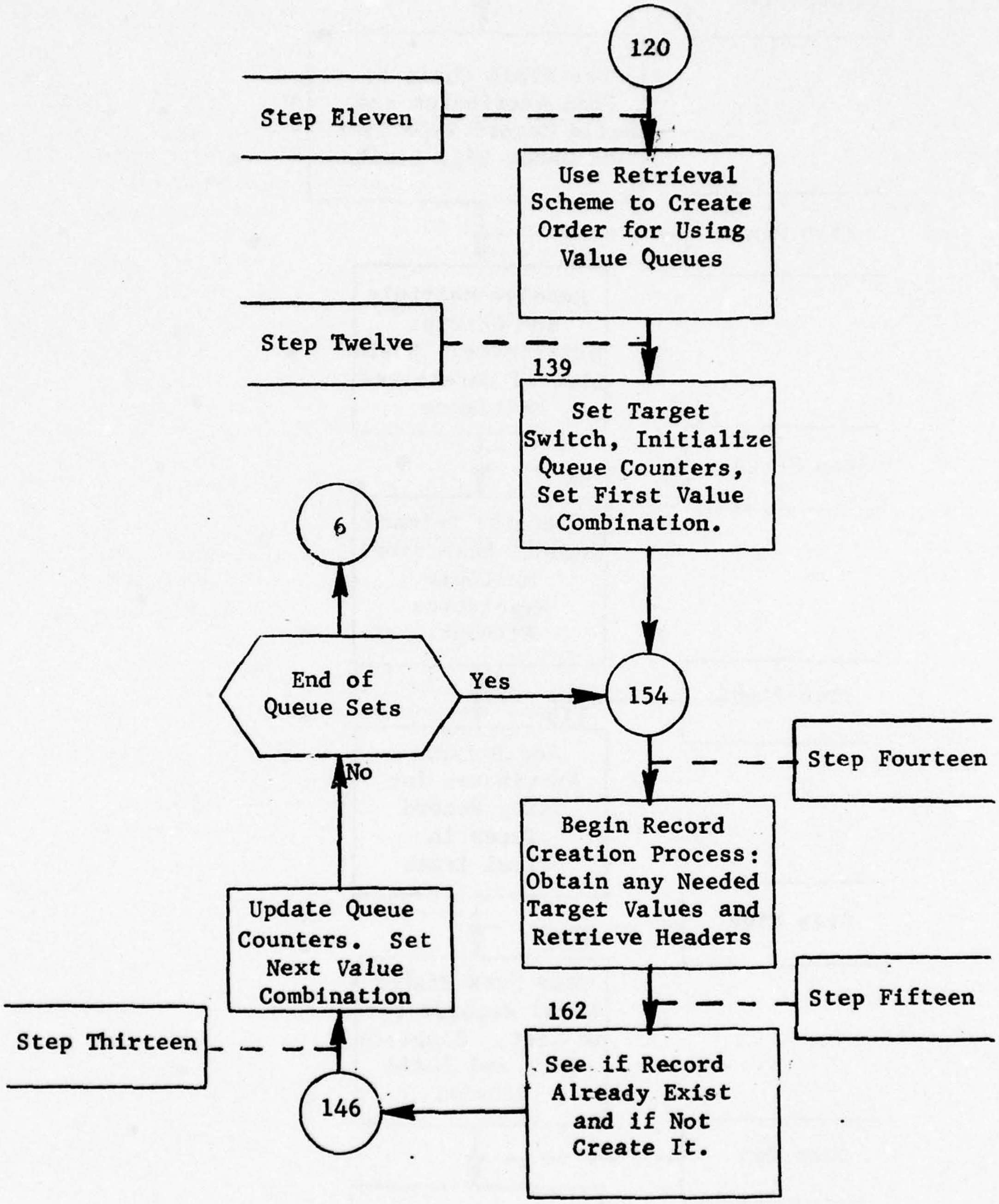



Figure 39. (Part 3 of 3)

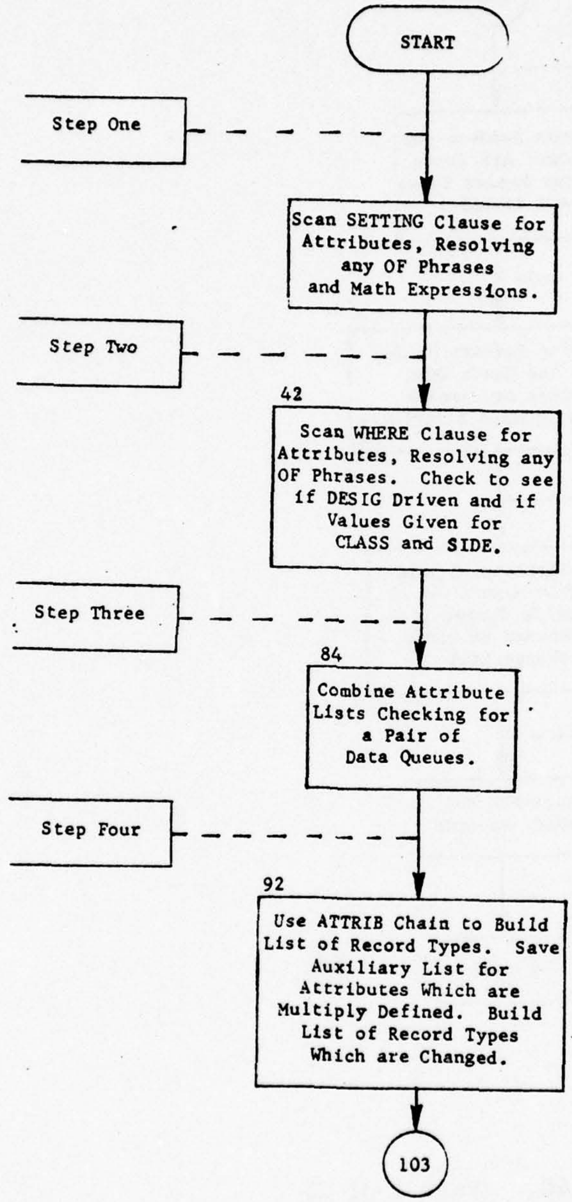



Figure 40. Subroutine CHANGE (Macro Flowchart)
(Part 1 of 2)

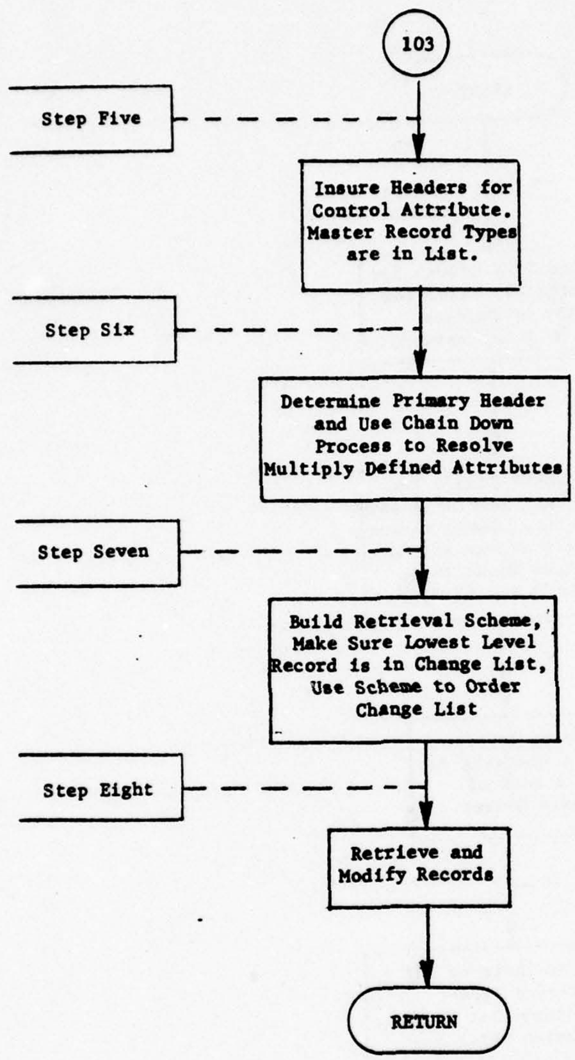



Figure 40. (Part 2 of 2)

Table 15. DATA Module Internal Common Blocks

| BLOCK | ARRAY OR VARIABLE | DESCRIPTION |
|---|---|---|
| ORDER | SCHORD(100) | Record type numbers of record types involved in retrieval scheme in retrieval order |
| | SORDNM(100) | Record type names in retrieval scheme order |
| | LENSCH | Length, in words, of retrieval scheme |
| PRINSP | PRINON | Switch to control optional prints<br>True - produce prints<br>False - do not produce prints |
| SCHEME | POINT | Pointers to current instruction of retrieval scheme |
| | SCHEME(200) | Retrieval scheme |
| SCRTCH | LIST(300) | Storage space used as work area by several subroutines |

4.7 Subroutine ENTMOD

| | |
|---|---|
| PURPOSE: | Entry subroutine for DATA module |
| ENTRY POINTS: | ENTMOD (first subroutine called when overlay DATA is executed) |
| FORMAL PARAMETERS: | None |
| COMMON BLOCKS: | PRINSP |
| SUBROUTINES CALLED: | CHANGE, CREAAT, DELETE, INSGET |
| CALLED BY: | MODGET |

Method:

This subroutine performs the function of selecting the desired DATA module overlay. INSGET is used to obtain the value of the verb which caused the call. The adverbs are scanned for the ONPRINTS adverb. If it is found the PRINON variable in the PRINSP common block is set to true. Then the requested overlay link is read in and the verb executed.

Subroutine ENTMOD (DATA) is illustrated in figure 41.

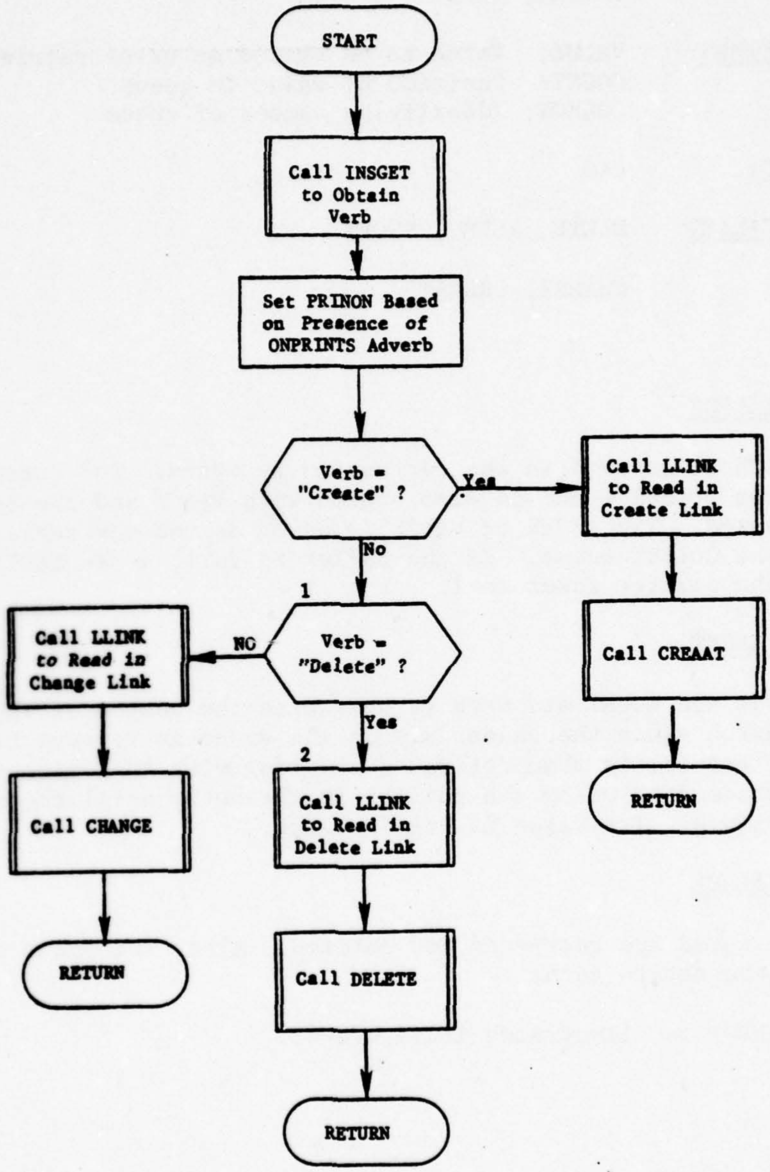



Figure 41. Subroutine ENTMOD (DATA)

## 4.7.1 Subroutine VALPUT*

PURPOSE: Store values for data queues

ENTRY POINTS: VALDEL, VALGET, VALPUT

FORMAL PARAMETERS:
VALUE: Value to be stored on value retrieved
COUNT: Position of value in queue
COLECT: Identifying number of queue

COMMON BLOCKS: C40

SUBROUTINES CALLED: DLETE, RETRV, STORE

CALLED BY: CHANGE, CREAAT

Method:

Entry Point VALPUT

The input VALUE is entered in the current entry space. The current pointer for the COLECT queue is also stored with VALUE and the queue count incremented. The index of VALUE is saved as the new current pointer for the COLECT queue. If the buffer is full, a new TABLEZ is created and the pointer reset to 1.

Entry Point VALGET

The queue count and COUNT are used to determine the number of entries needed for search since the values are in the queue in reverse order. The number of entries is then retrieved starting with the current pointer and proceeding using the pointer in the entry until the desired entry is retrieved. Its value is set in VALUE.

Entry Point VALDEL

Any TABLEZs created are retrieved and deleted. Also, the queue counts and pointers are set to zero.

Subroutine VALPUT is illustrated in figure 42.

*This subroutine appears in overlays DATACH and DATACR.

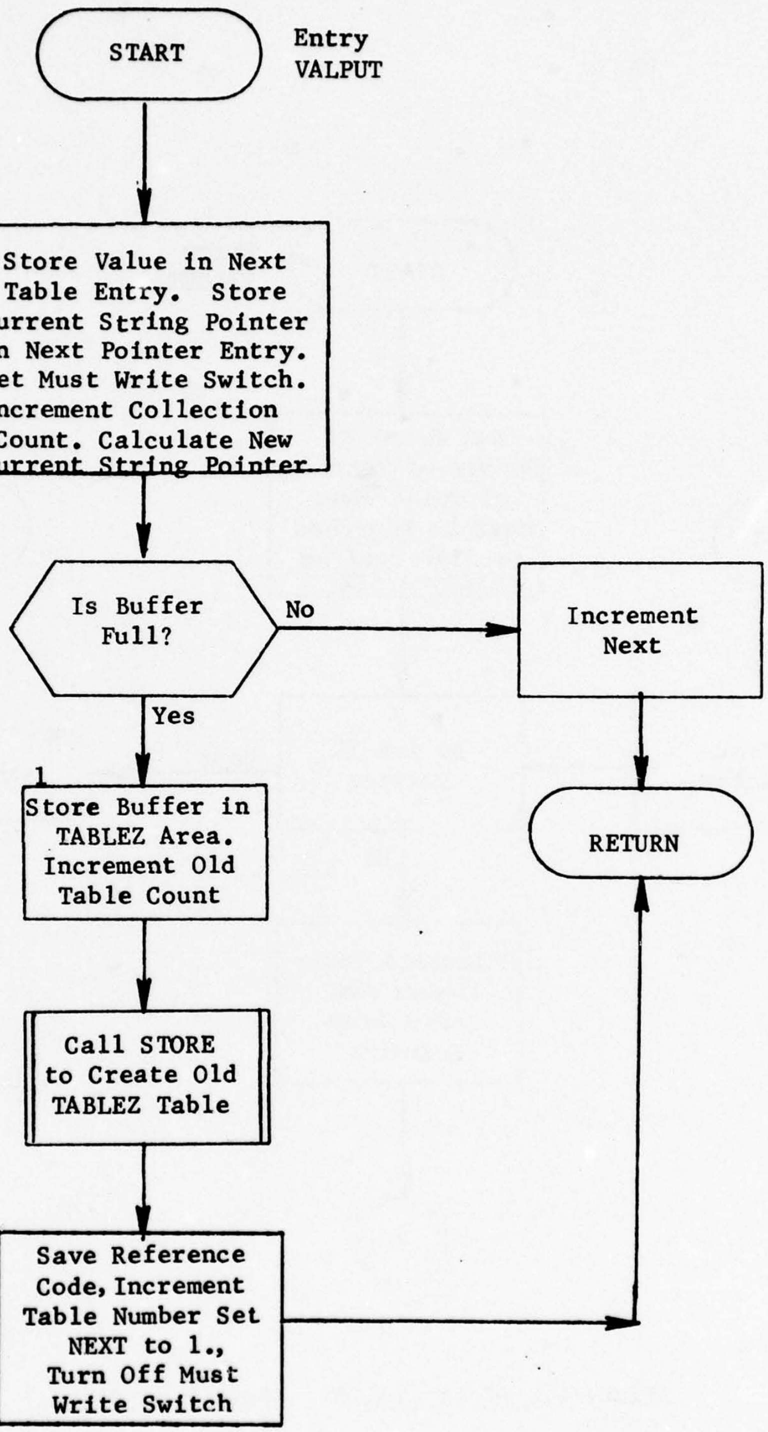



Figure 42. Subroutine VALPUT: Entry VALPUT (Part 1 of 4)

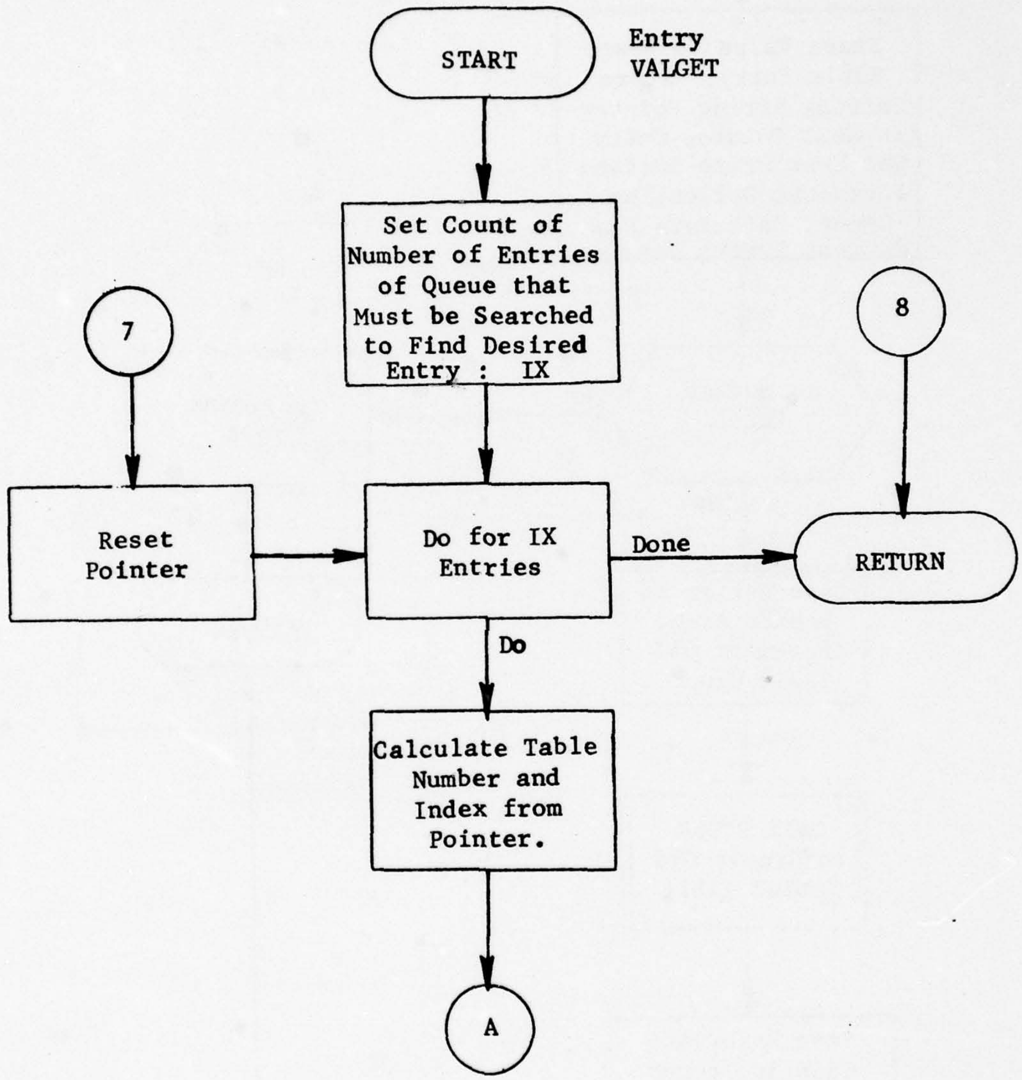



Figure 42. Entry VALGET (Part 2 of 4)

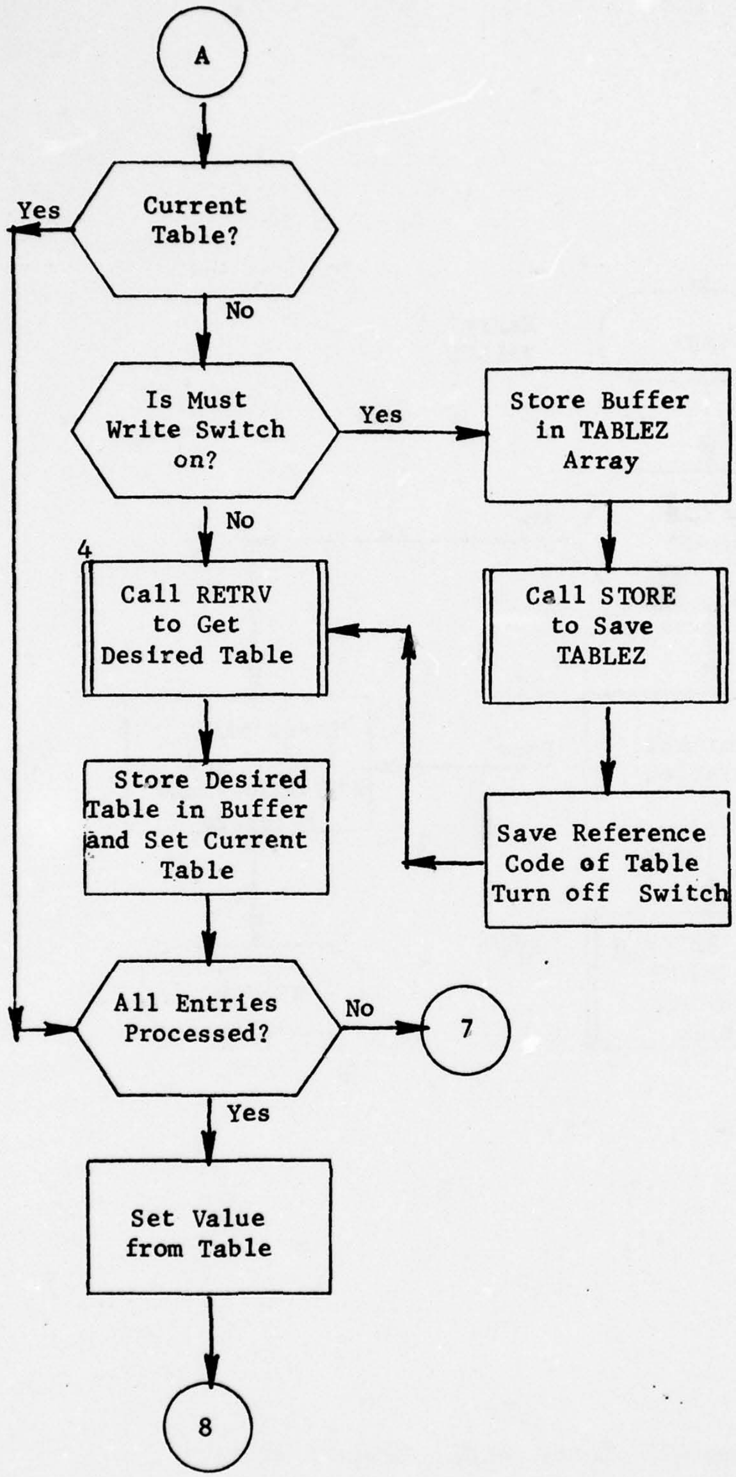



Figure 42. Entry VALGET (Part 3 of 4)

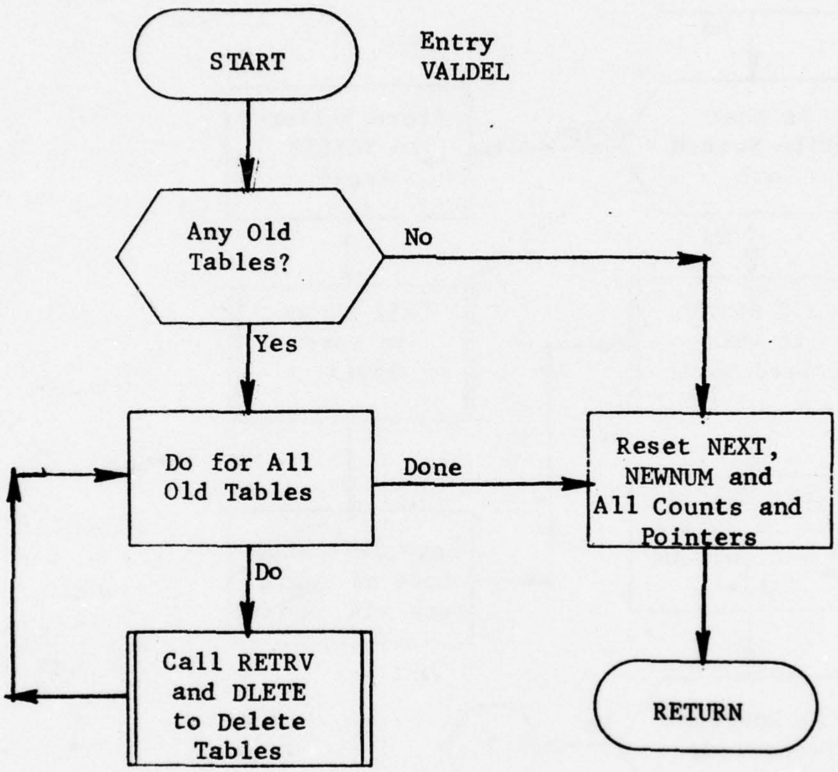



Figure 42. Entry VALDEL (Part 4 of 4)

## 4.8 Subroutine CHANGE*

PURPOSE: To change existing records

ENTRY POINTS: CHANGE

FORMAL PARAMETERS: None

COMMON BLOCKS: C10, C20, C30, ERRCOM, OOPS, ORDER, PRINSP, SCHEME

SUBROUTINES CALLED: DESSCH, GETNXT, HDFND, HEAD, INSGET, LINKUP, MODFY, NEXTTT, NXTDES, OFVAL, PRIMHD, SETSCH, UNCODE, VALDEL, VALFND, VALGET, VALPUT, XMATH, XWHERE

CALLED BY: ENTMOD (DATA)

Method:

The CHANGE verb process may be broken into eight steps which are carried out in sequence (see figure 40).

Step One

The SETTING clause is scanned. In the process any OF phrases and LIKE strings are immediately resolved using VALFND and their values stored via OFVAL. Also the extent of any mathematical calculations is noted as the limits for execution for subroutine XMATH. The main thrust of the step is to list any attributes (ATNUMB) and save the values assigned to them. If an extended equals phrase is included, a queue of the values is built and the involved attribute(s) are flagged (ATTYPE=2) (see figure 43).

Step Two

The WHERE clause is scanned. In the process any OF phrases and LIKE strings are immediately resolved using VALFND and their values stored via OFVAL. Any attributes are saved in a list (WHATNB). If an extended equals phrase is included, its involved attributes are flagged (WHATYP=4) and its values stored in a queue via VALPUT. The CLASS and SIDE attributes are particularly noted and their values saved to assist in the retrieval scheme construction process. If the DESIG attribute is the only attribute included this fact is noted as the retrieval process differs greatly in this event (see figure 44).

*The main routine of overlay DATACH

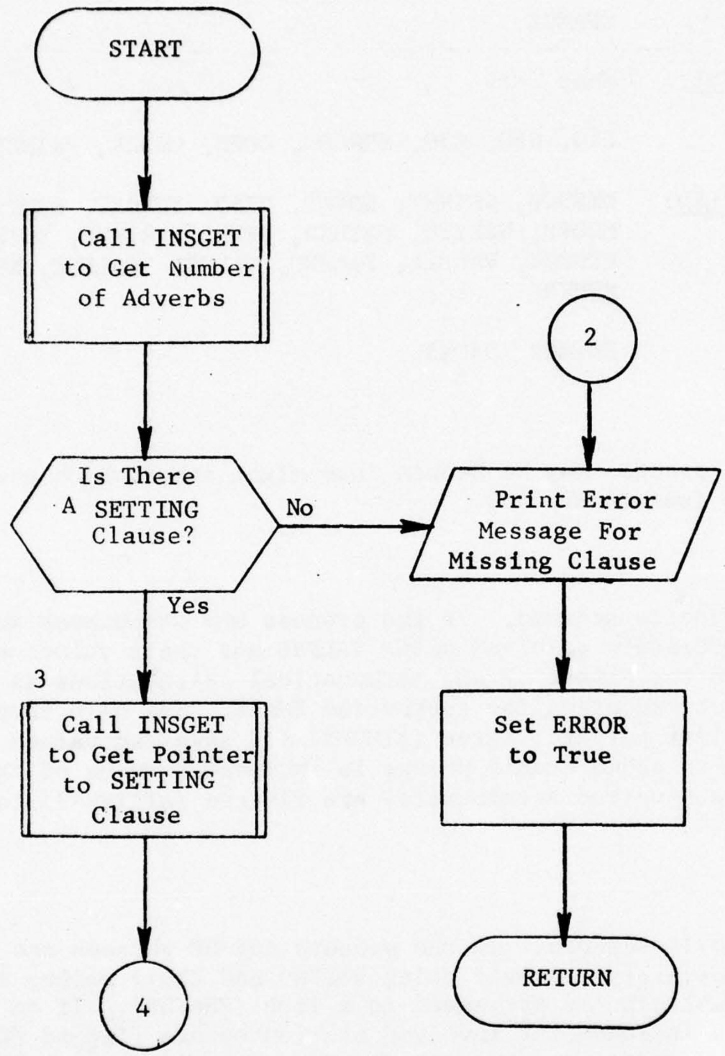



**Figure 43. Subroutine CHANGE: Step One (Part 1 of 10)**

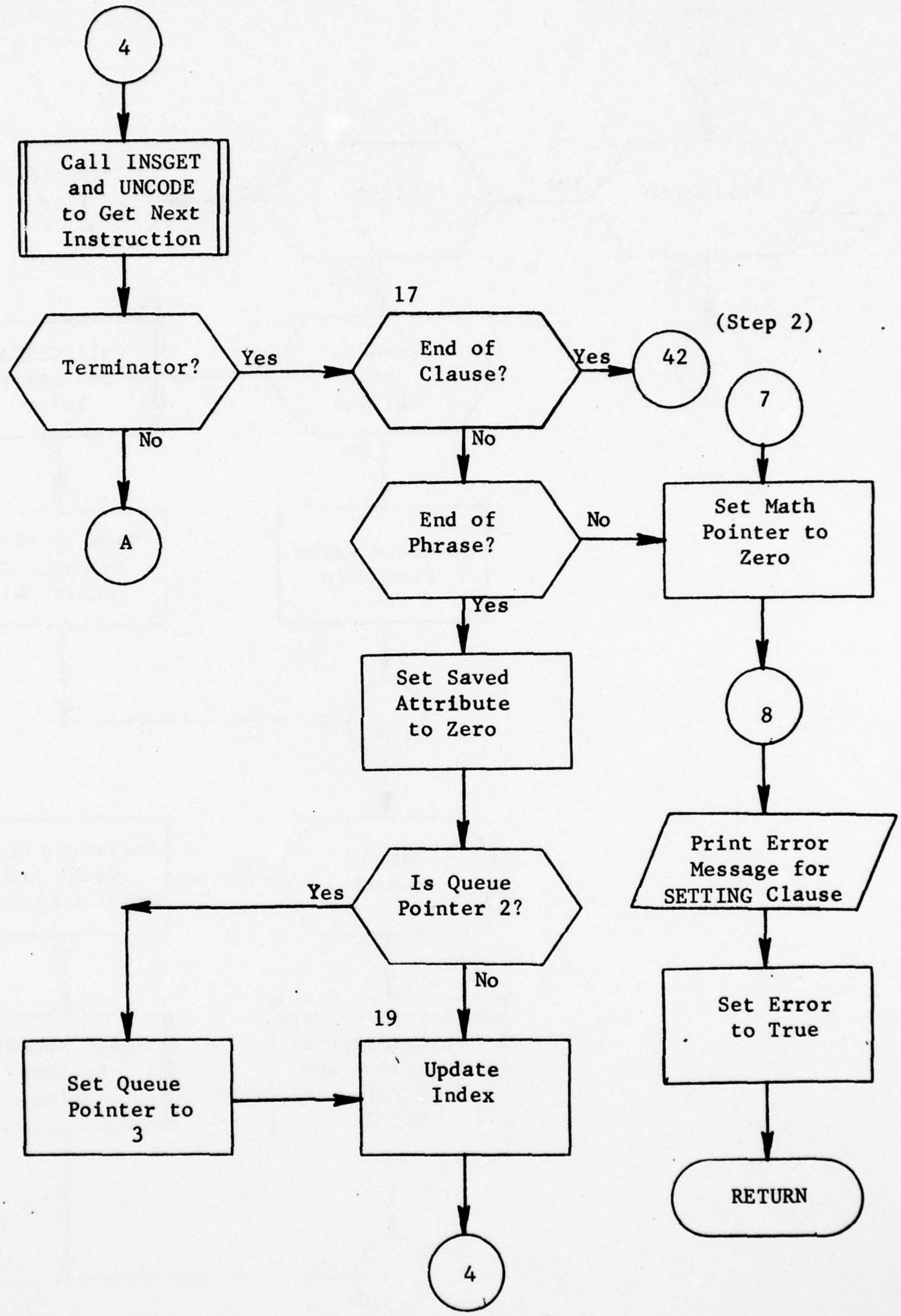



Figure 43. (Part 2 of 10)

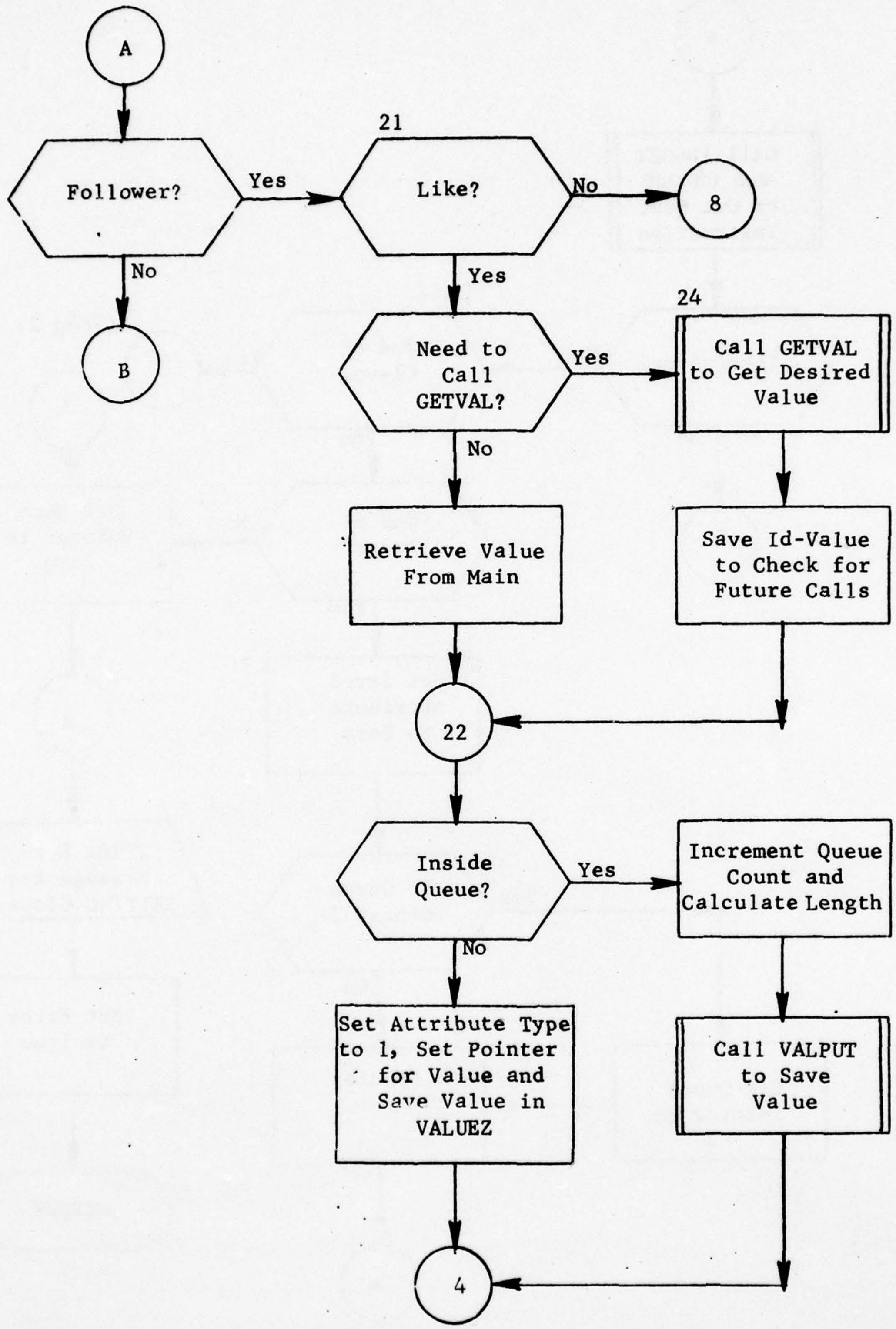



Figure 43. (Part 3 of 10)

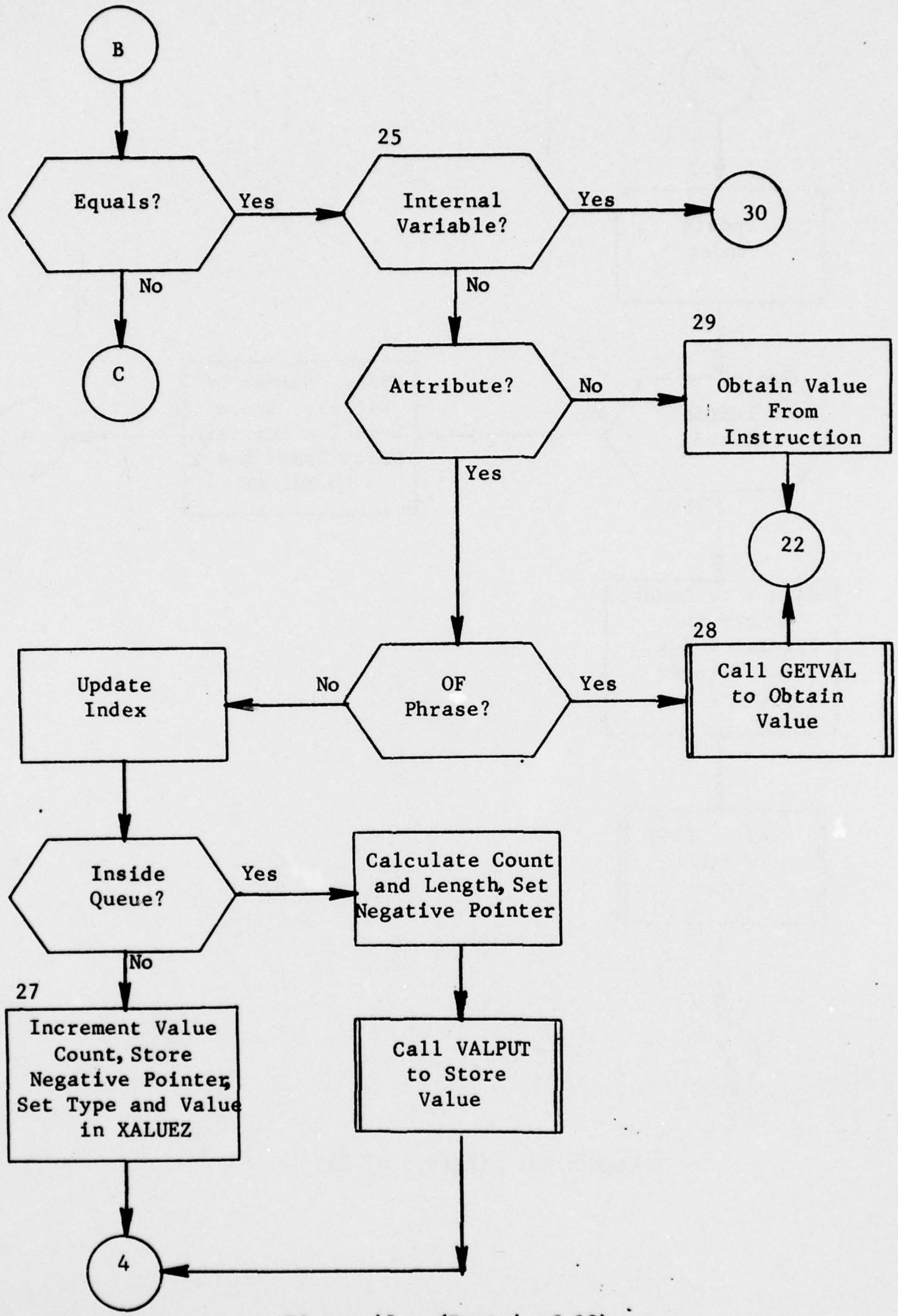



Figure 43. (Part 4 of 10)

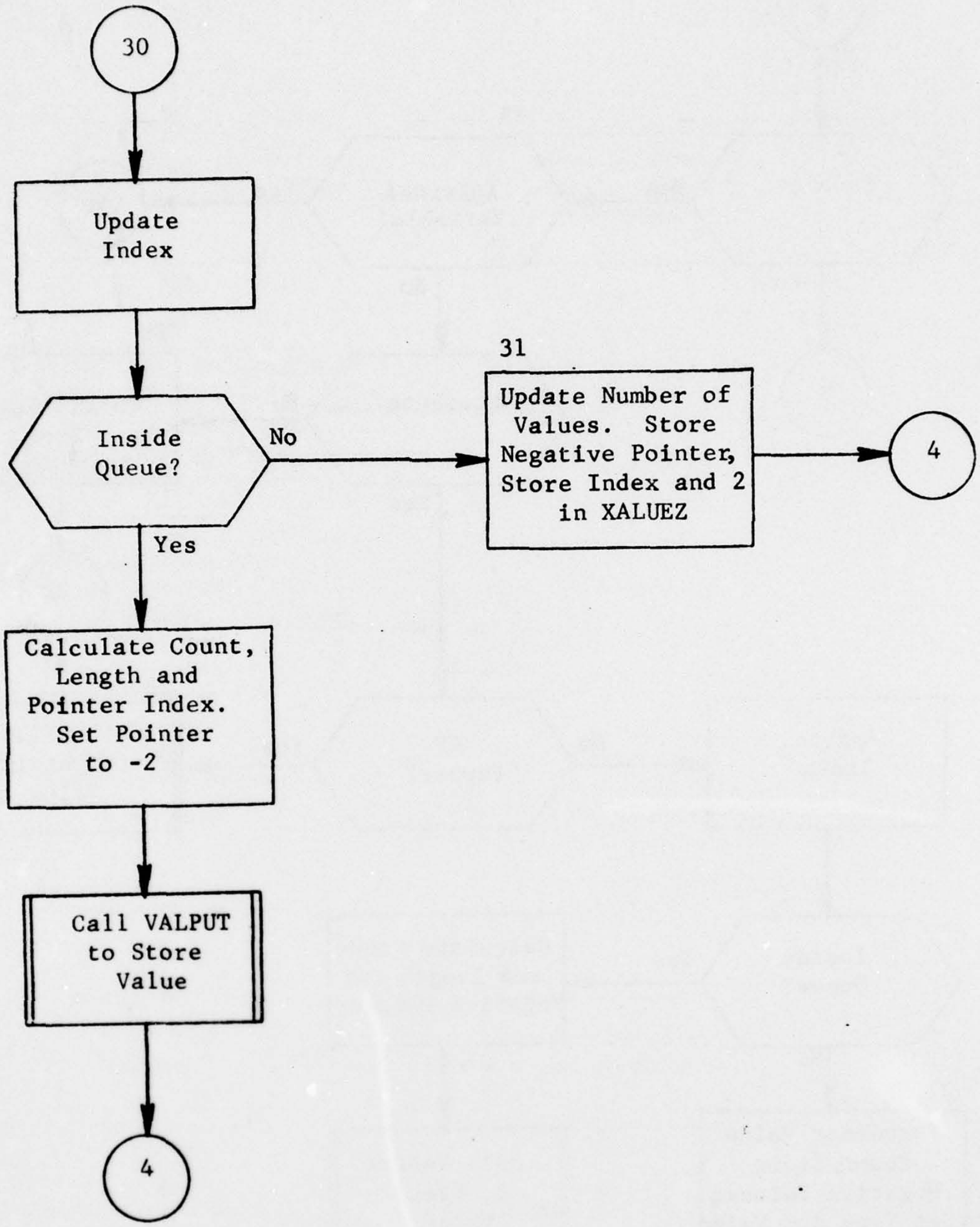



**Figure 43. (Part 5 of 10)**

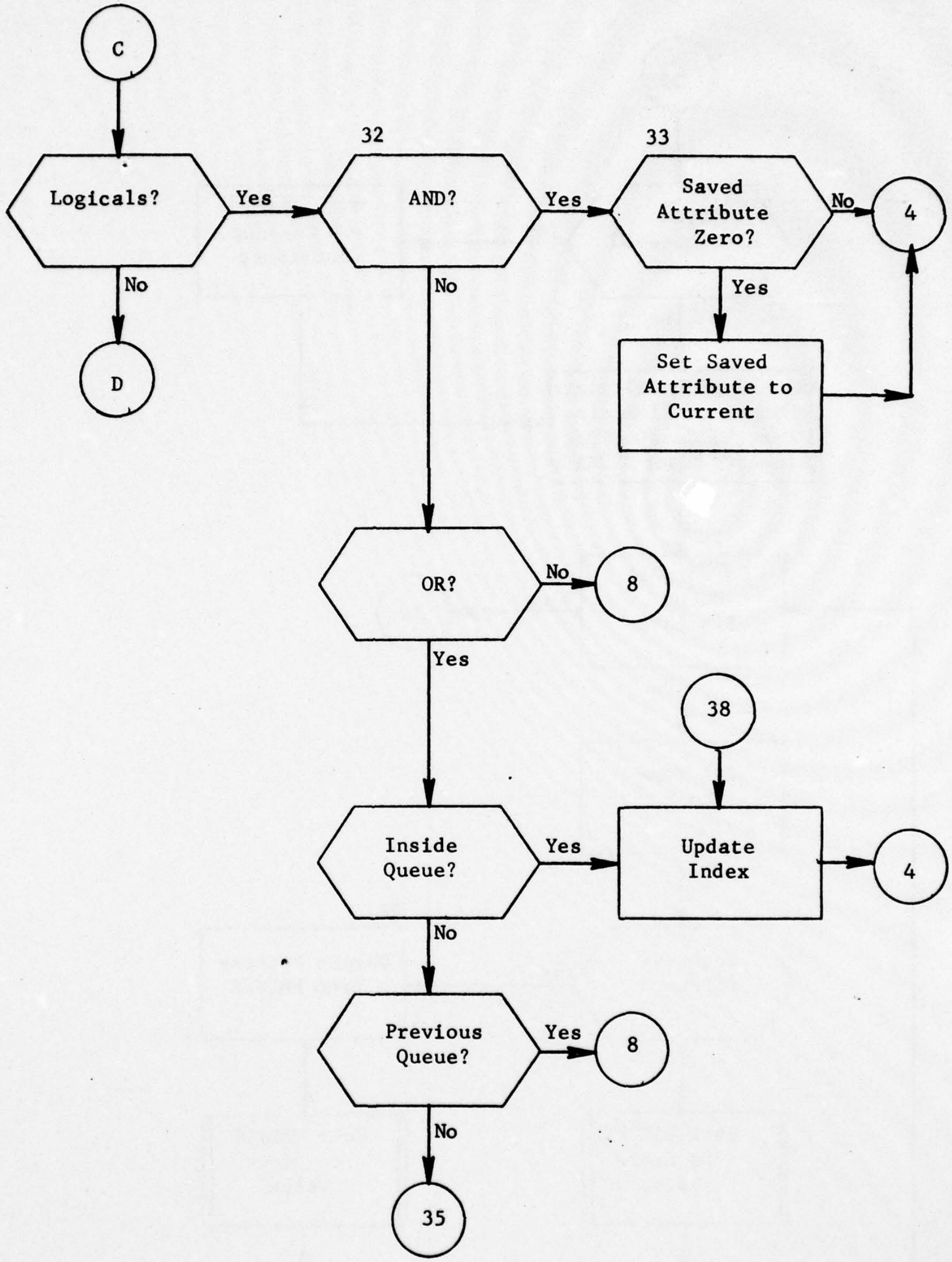



Figure 43. (Part 6 of 10)

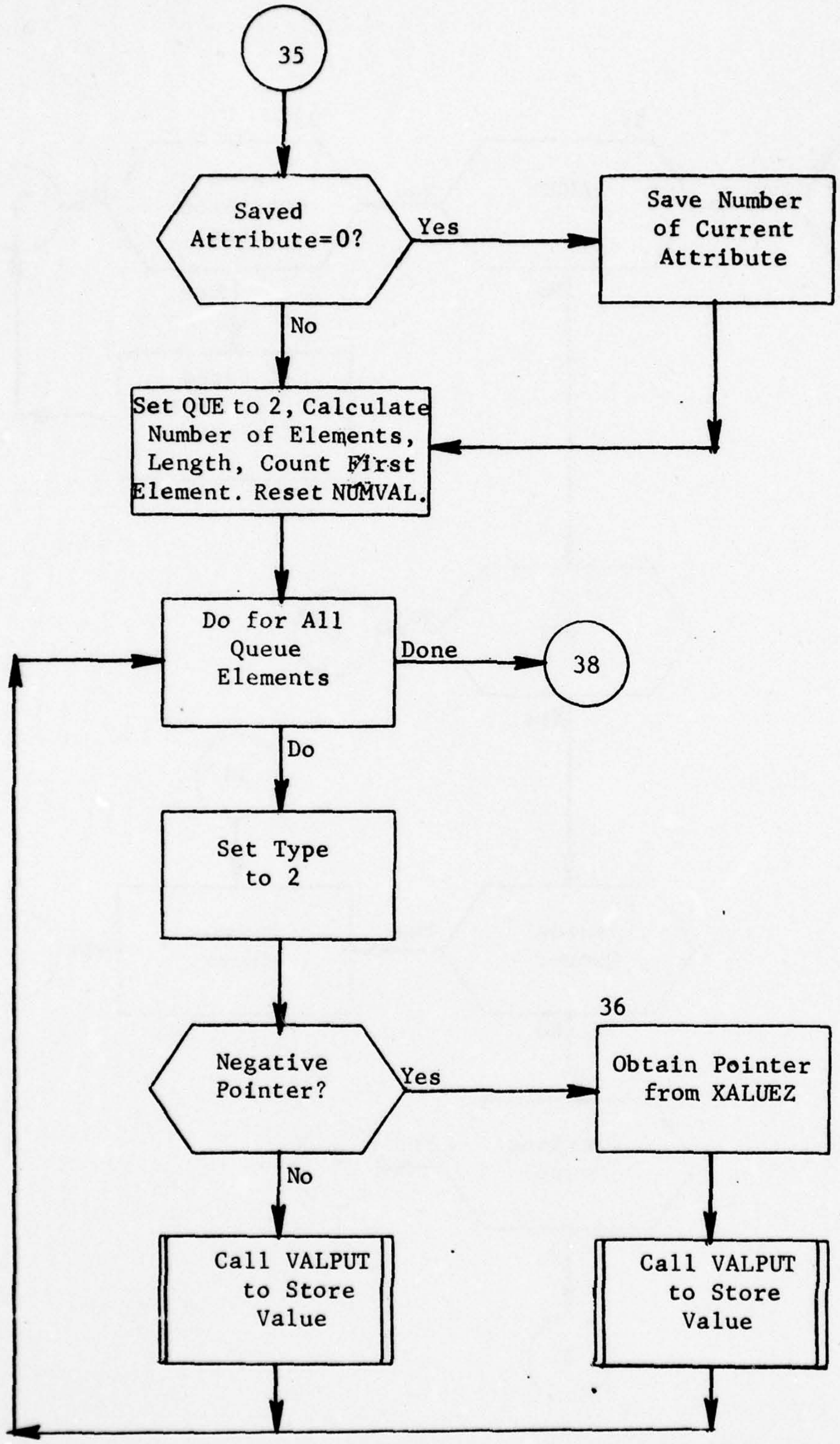



Figure 43. (Part 7 of 10)

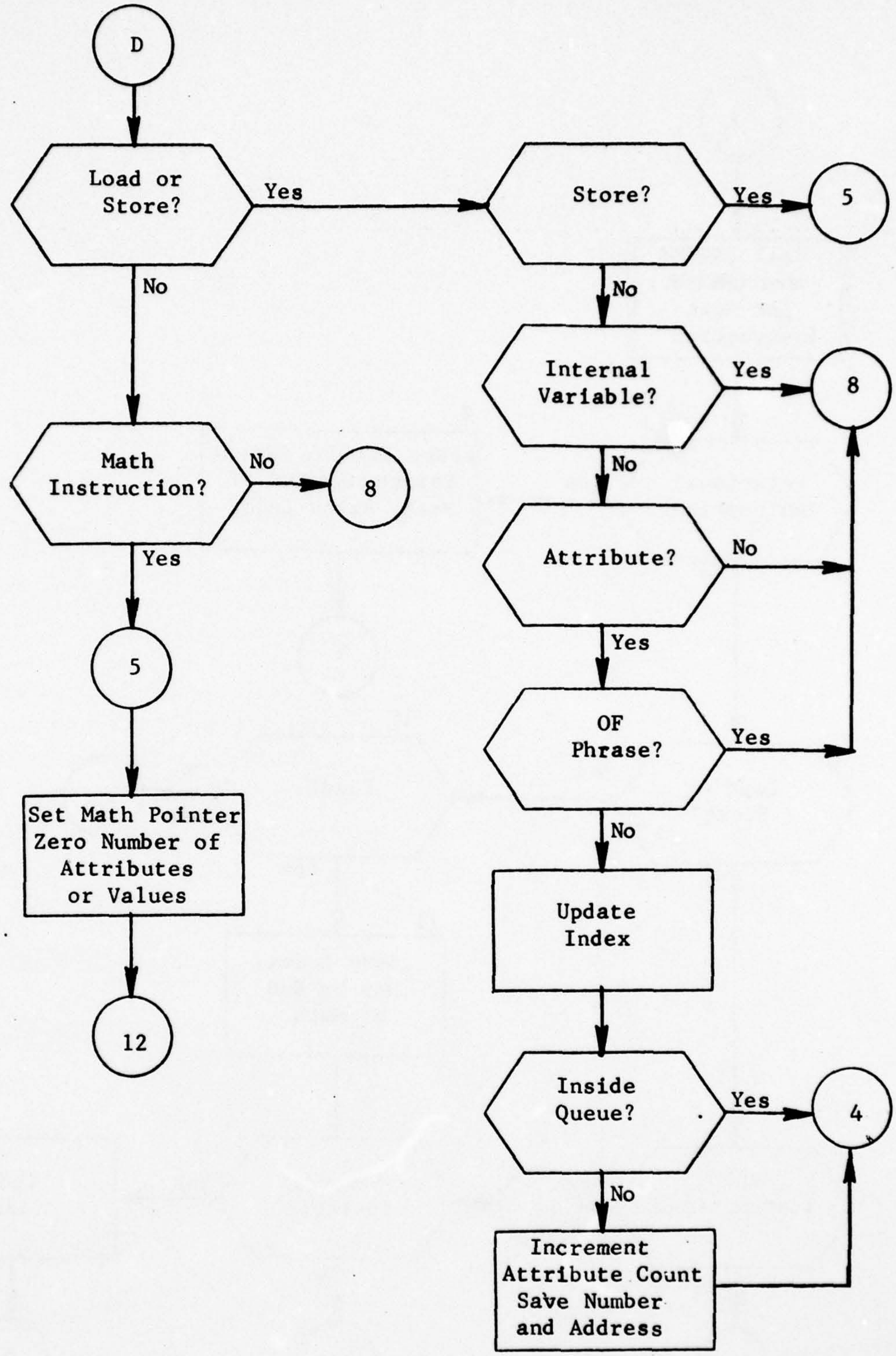



Figure 43. (Part 8 of 10)

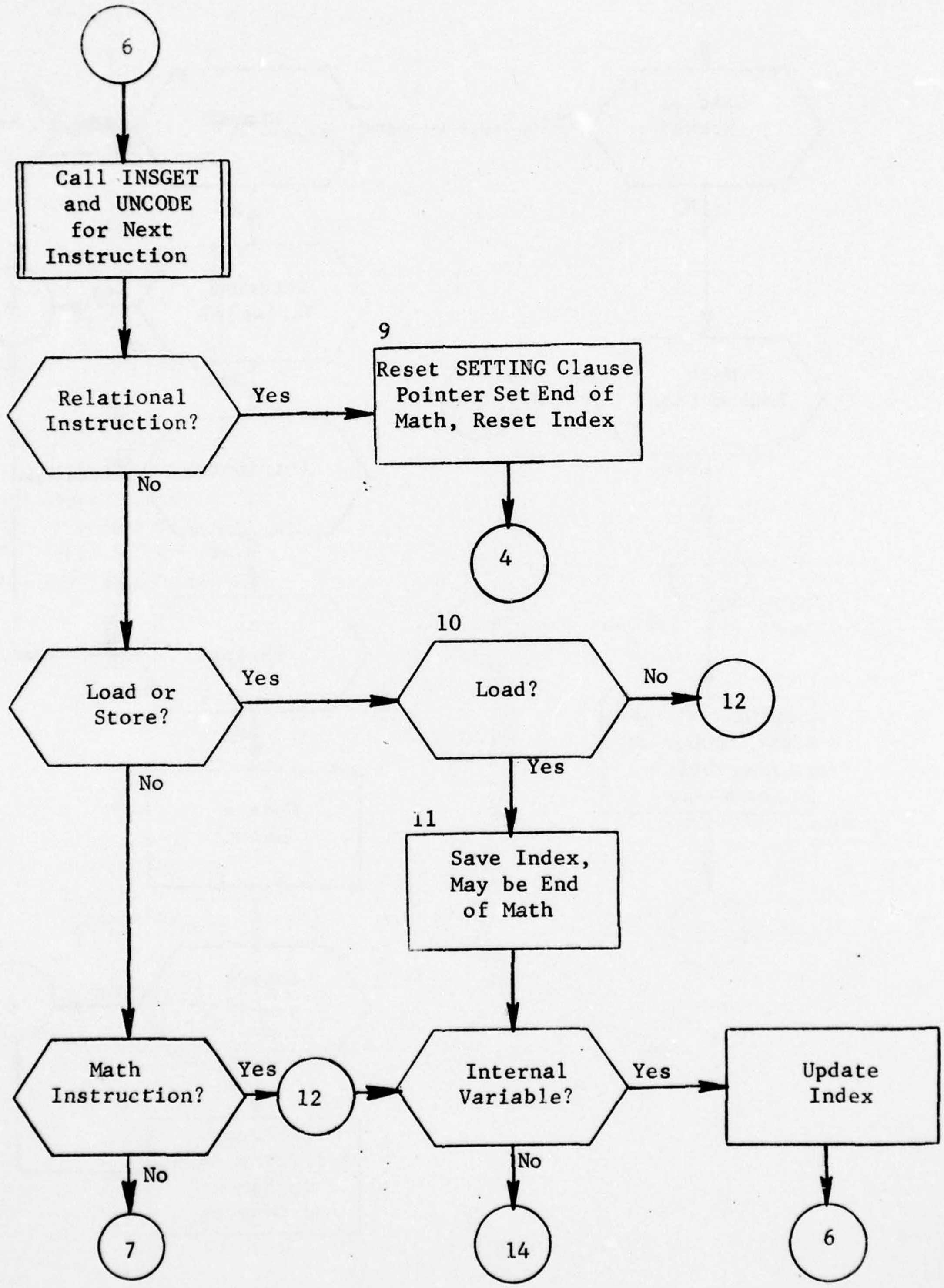



Figure 43. (Part 9 of 10)

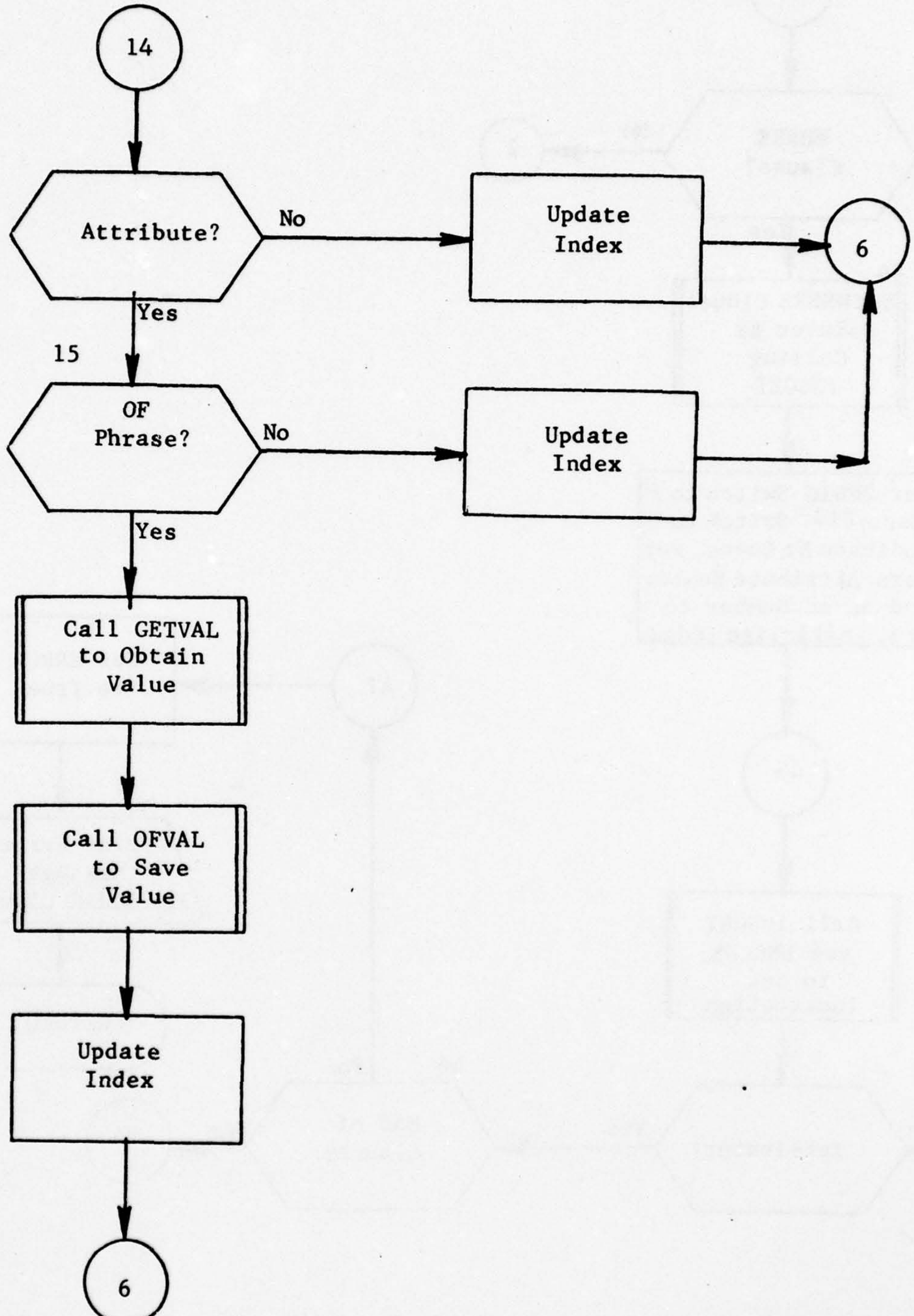



Figure 43. (Part 10 of 10)

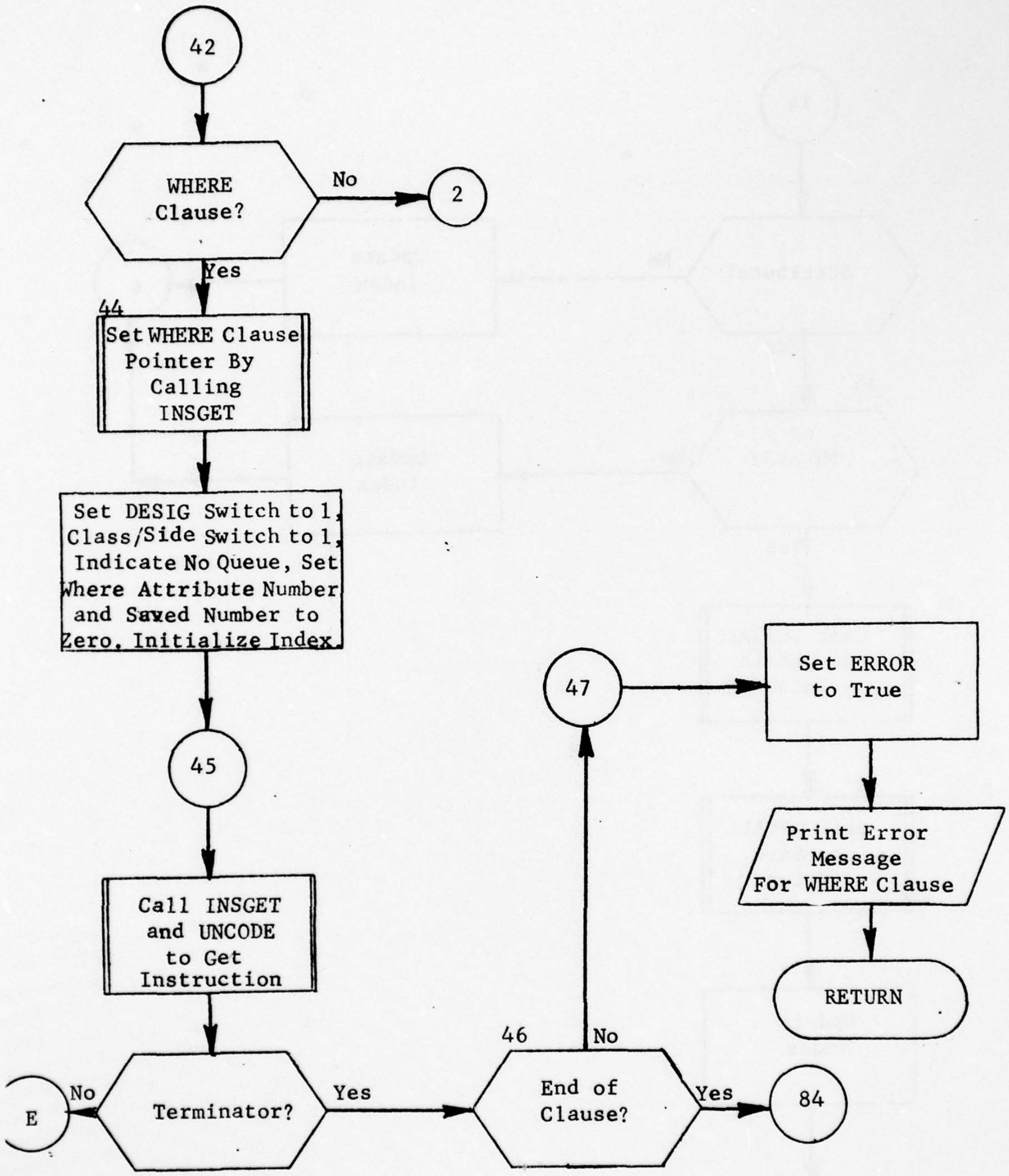



Figure 44. Subroutine CHANGE: Step Two (Part 1 of 9)

AD-A054 377 COMMAND AND CONTROL TECHNICAL CENTER WASHINGTON D C F/G 15/7
THE CCTC QUICK-REACTING GENERAL WAR GAMING SYSTEM. (QUICK). PRO--ETC(U)
JUN 77 D J SANDERS, P F MAYKRANTZ, J M HERRIN

UNCLASSIFIED CCTC-CSM-MM-9-77-V1-PT-1 SBIE-AD-E100 051 NL

4 OF 5
AD
A054377

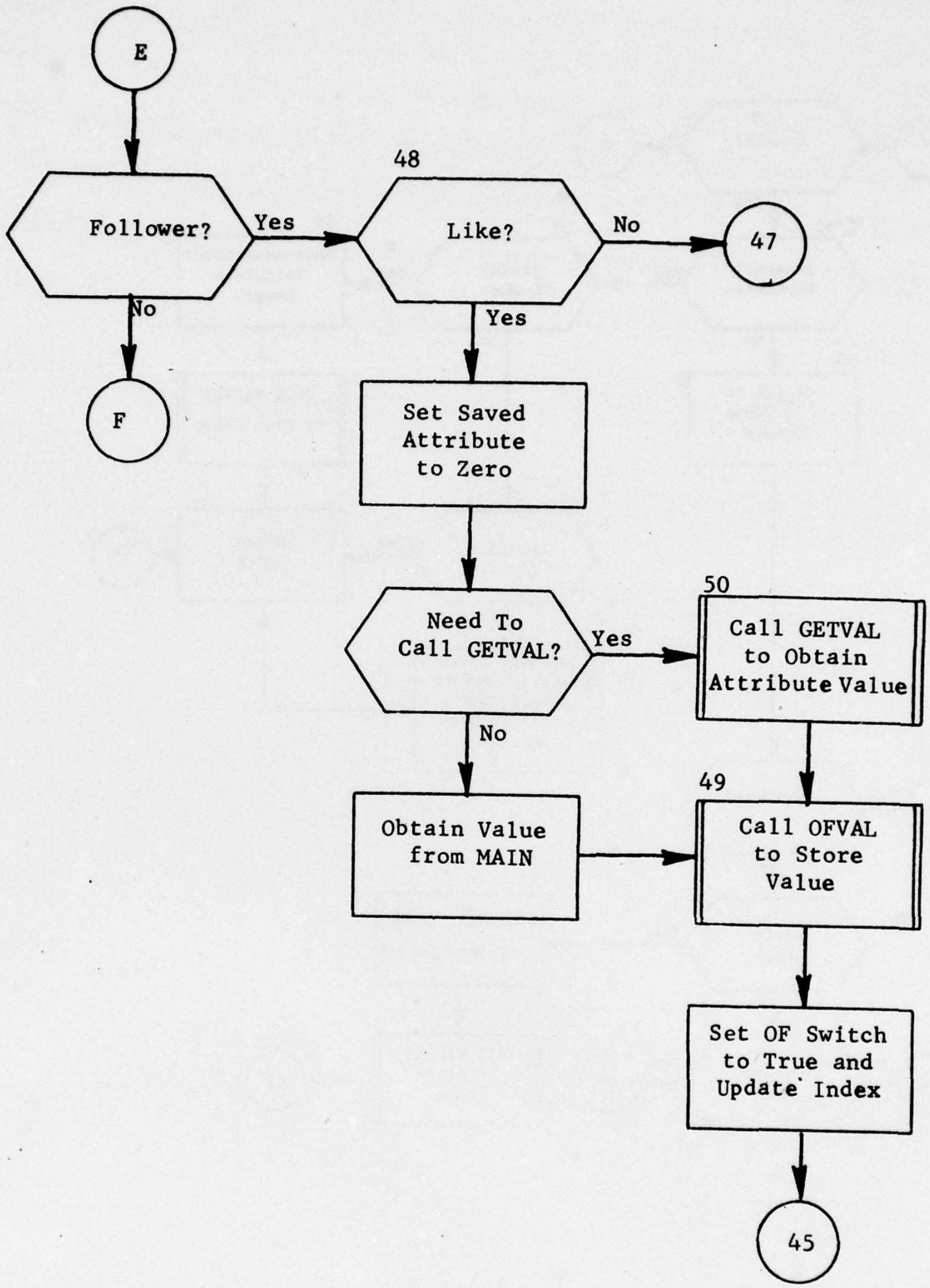



Figure 44. (Part 2 of 9)

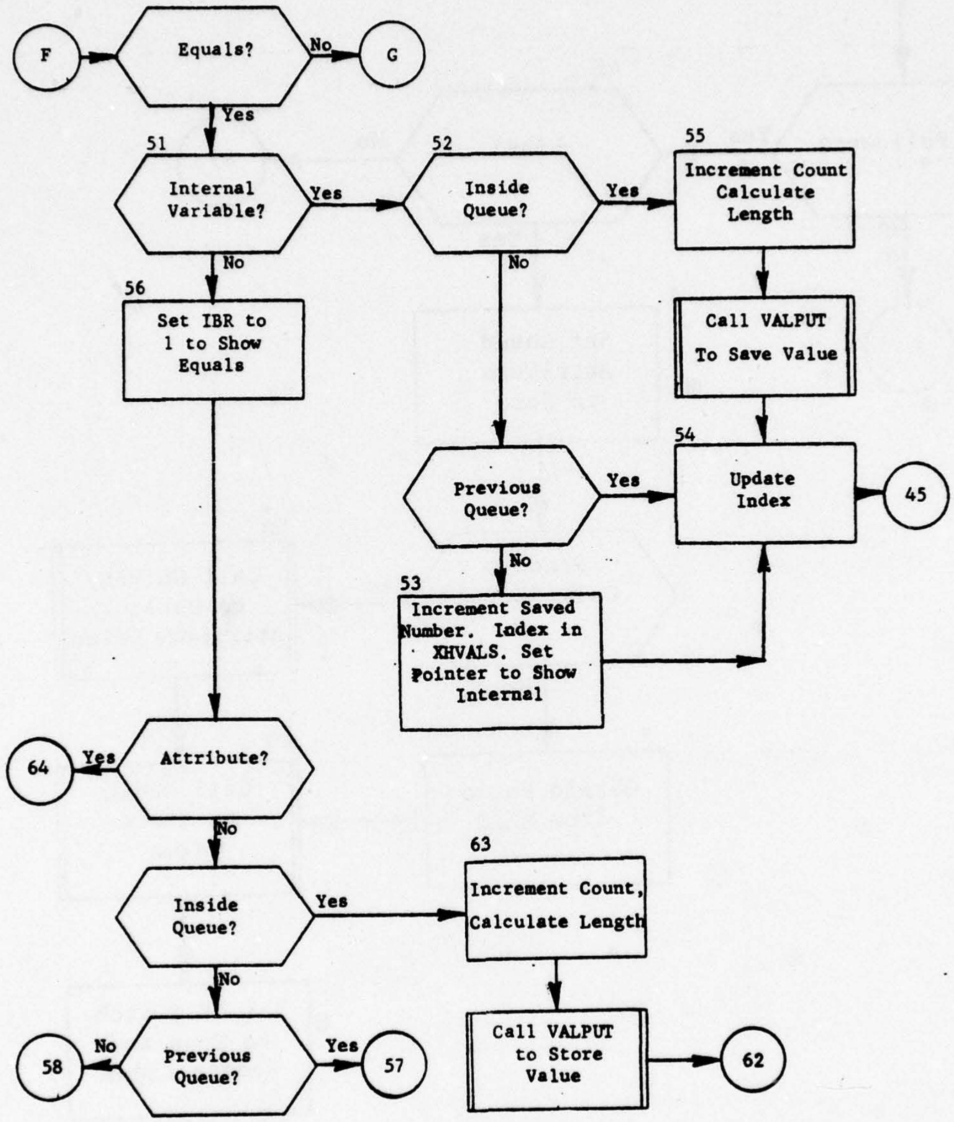



Figure 44. (Part 3 of 9)

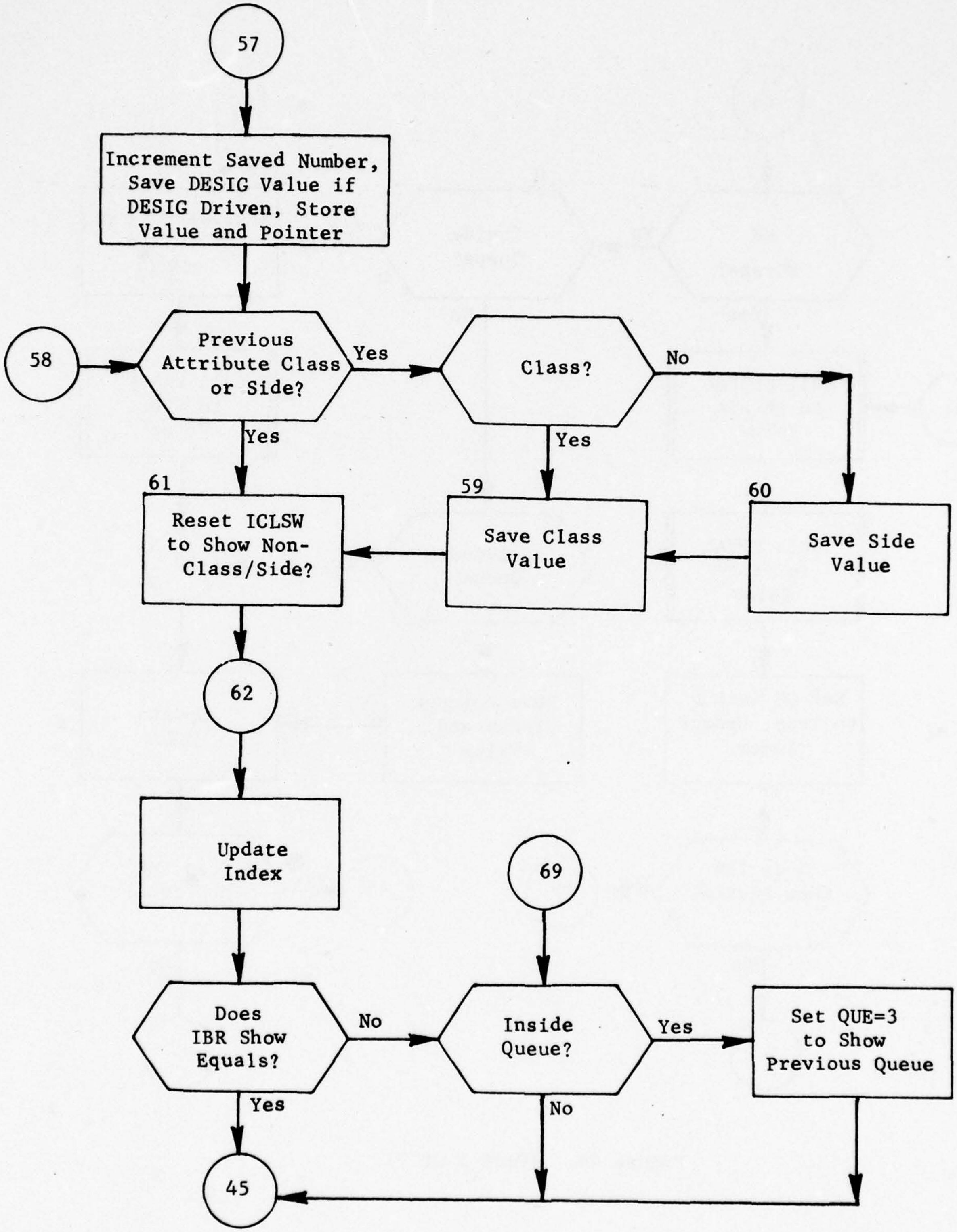



Figure 44. (Part 4 of 9)

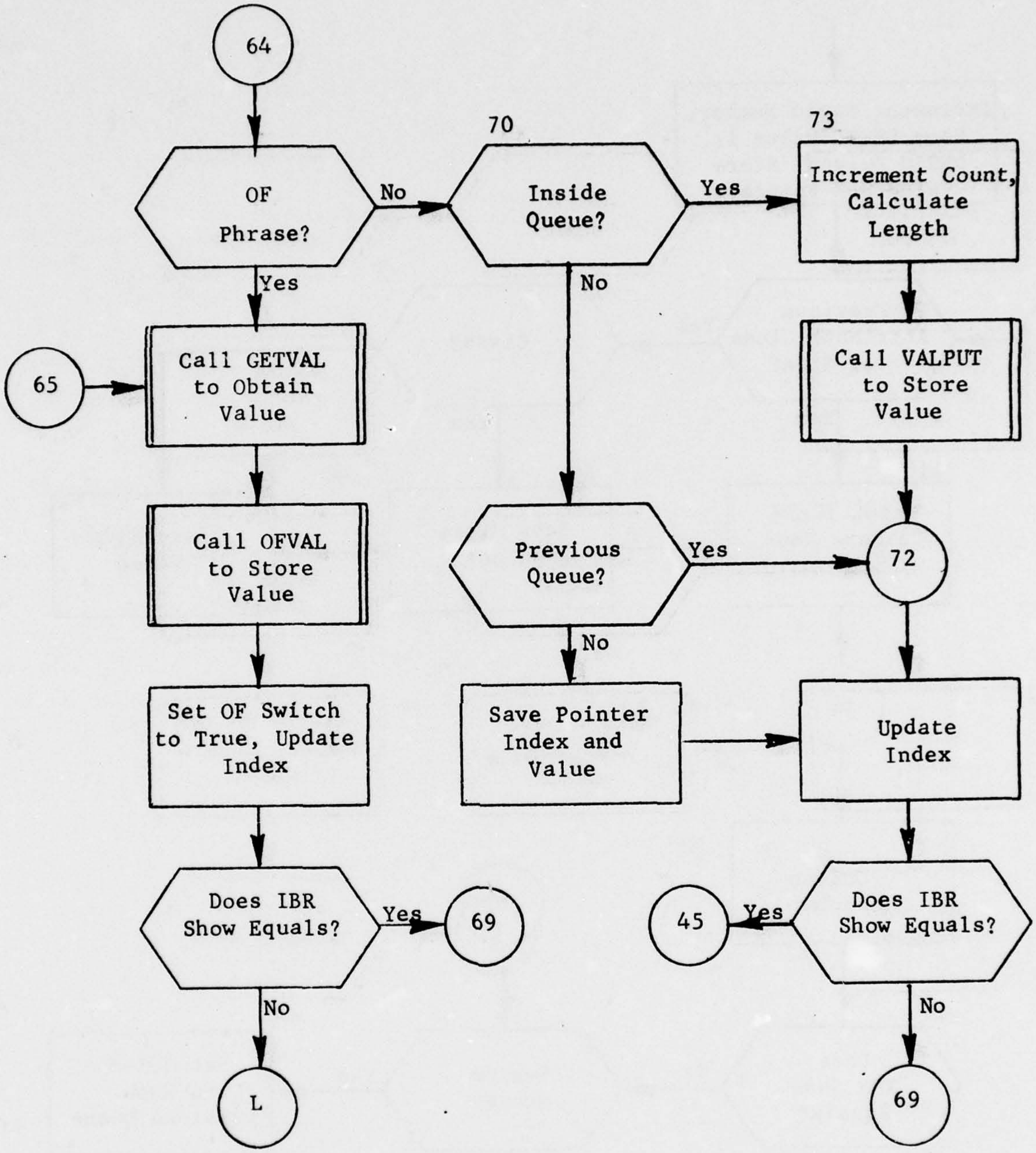



Figure 44. (Part 5 of 9)

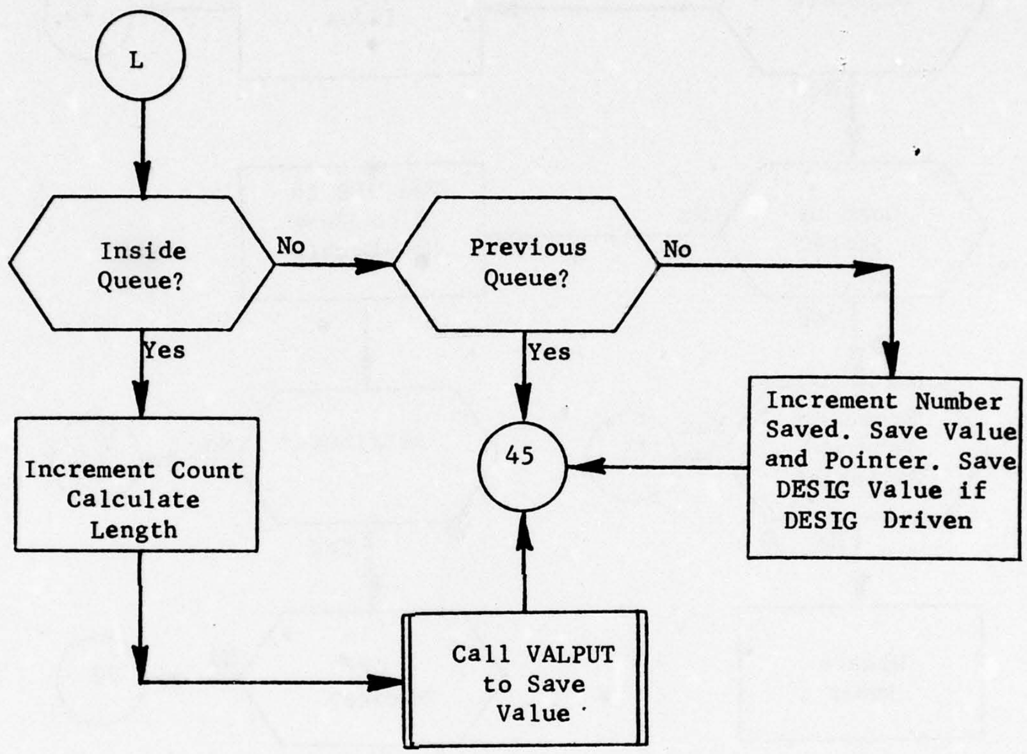



Figure 44. (Part 6 of 9)

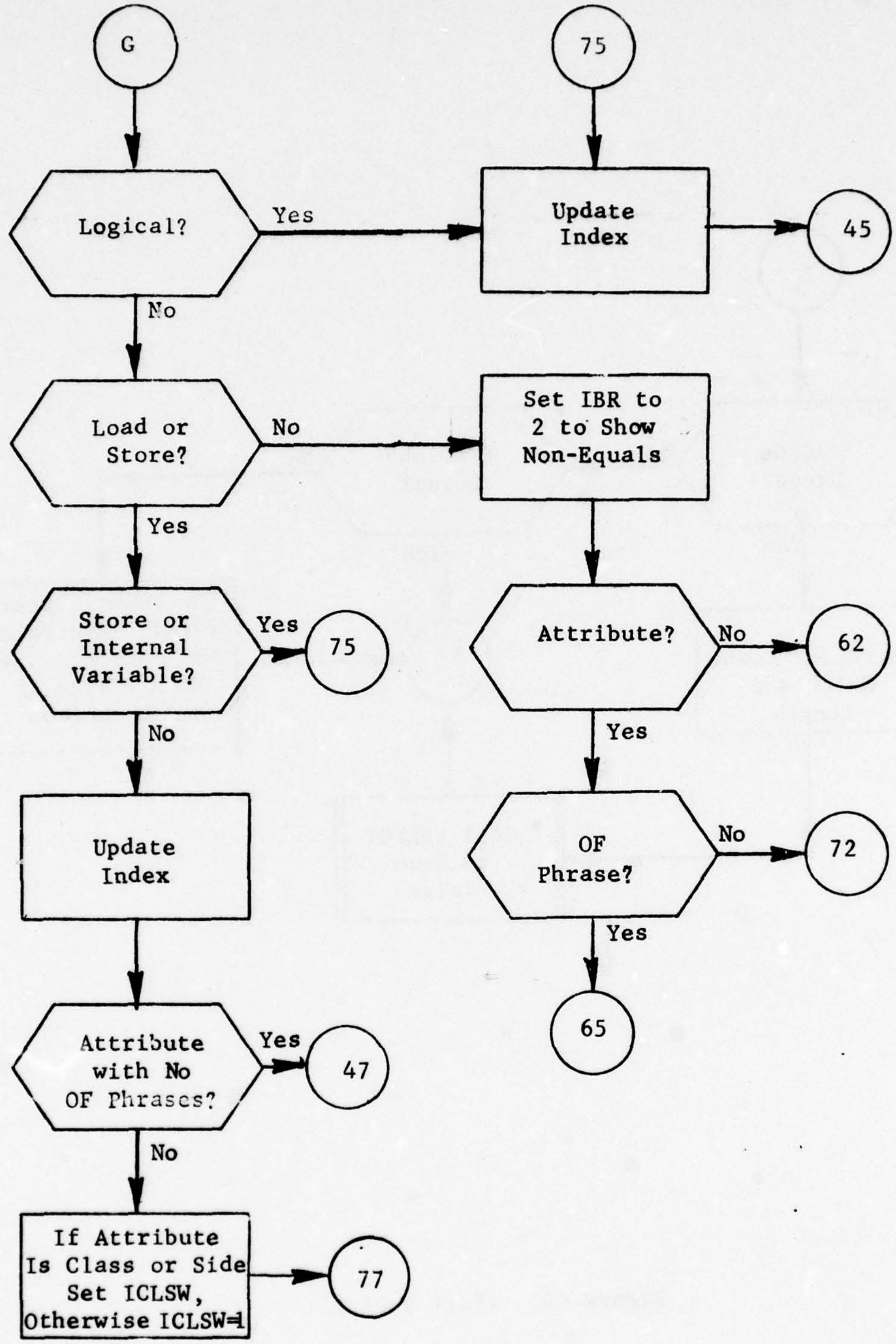



Figure 44. (Part 7 of 9)

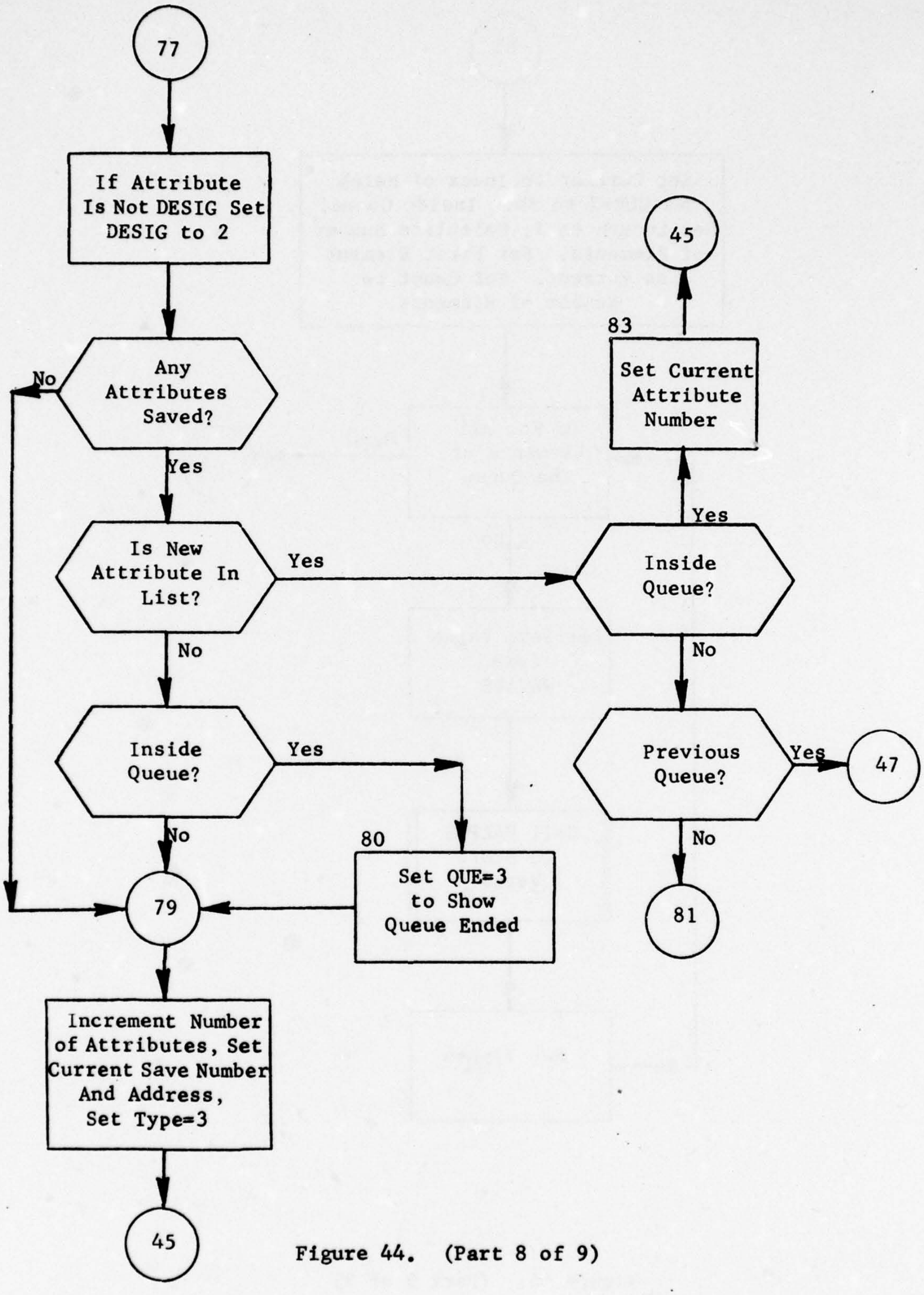



Figure 44. (Part 8 of 9)

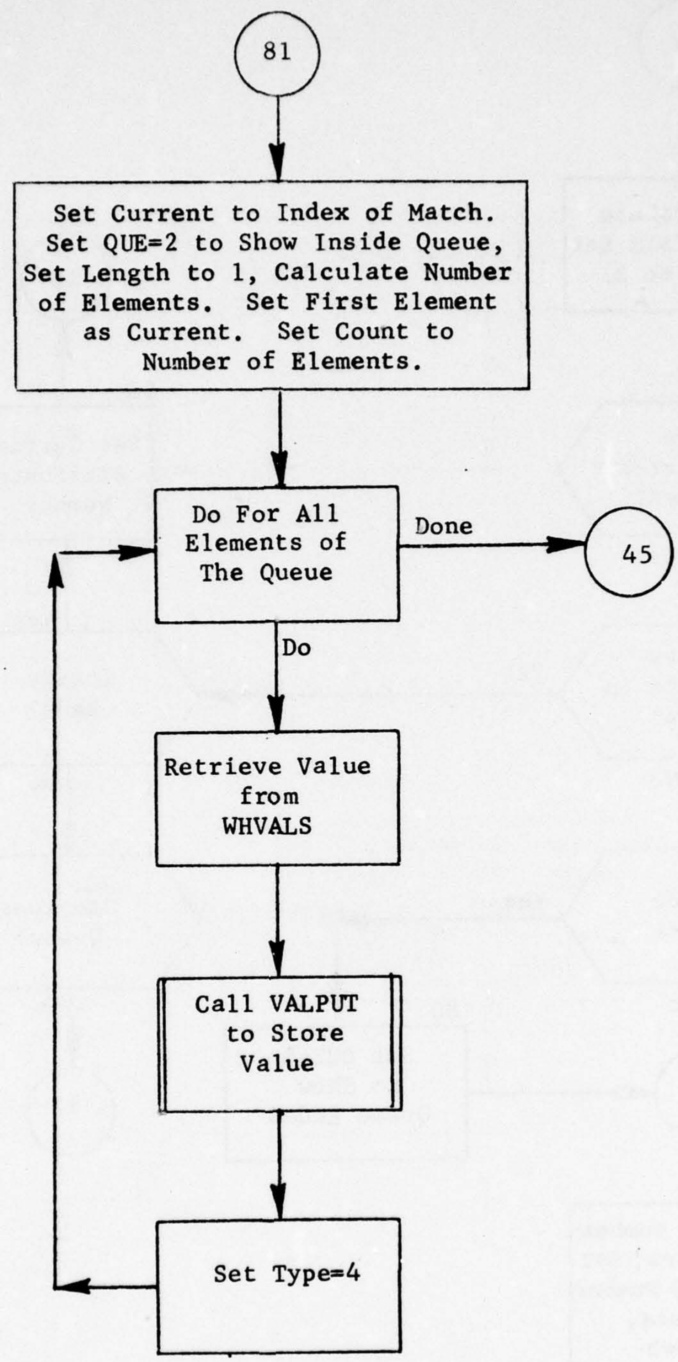



Figure 44. (Part 9 of 9)

## Step Three

The two attributes lists (ATNUMB and WHATNB) are now combined. Also it is determined if both clauses contained a queued set of attributes (see figure 45).

## Step Four

The list of attributes (ATNUMB) is now used in an attempt to build a list of record types. The ATRIB chain is used to find the attributes in the list and it is determined for each attribute whether it is a single, multiple or control. For a single attribute, the record type it is on is added to the record type list (RTLIST). For multiple attributes (those in the SETTING clause) a list (MLTLST) is kept separately of the record types which contain them. For a control attribute the controlled record is added to the record type list (RTLIST). If the attribute appeared in the SETTING clause, the record type is also added to a special list (CTLLST). A separate list (CRECNM) is also made of the record types whose attributes appear in the SETTING clause as these are the types which will be changed (see figure 46).

## Step Five

For each record type in the list (CTLIST) of controlled record types whose attribute was in the SETTING clause, PRIMHD is now called to determine their primary header. These headers are added to the list of record types (RTLIST) (see figure 47).

## Step Six

Now the primary header is determined. If a value for CLASS was included, it is used to do this. If no such value was included the record type with the highest number is used. PRIMHD is called to determine the primary header. The primary chains down from the primary header are now checked against the multiple attribute record list. Any found are included in the record type list (RTLIST) (see figure 48).

## Step Seven

First LINKUP is called to complete the record type list (RTLIST). The lowest level record in the list is now checked to be sure it is in the list of record types to be changed (CRECNM). SETSCH is now called to build the retrieval scheme. The retrieval scheme is used to determine the retrieval order of the record types which are to be changed and they are placed in the CHORD list in this order (see figure 49).

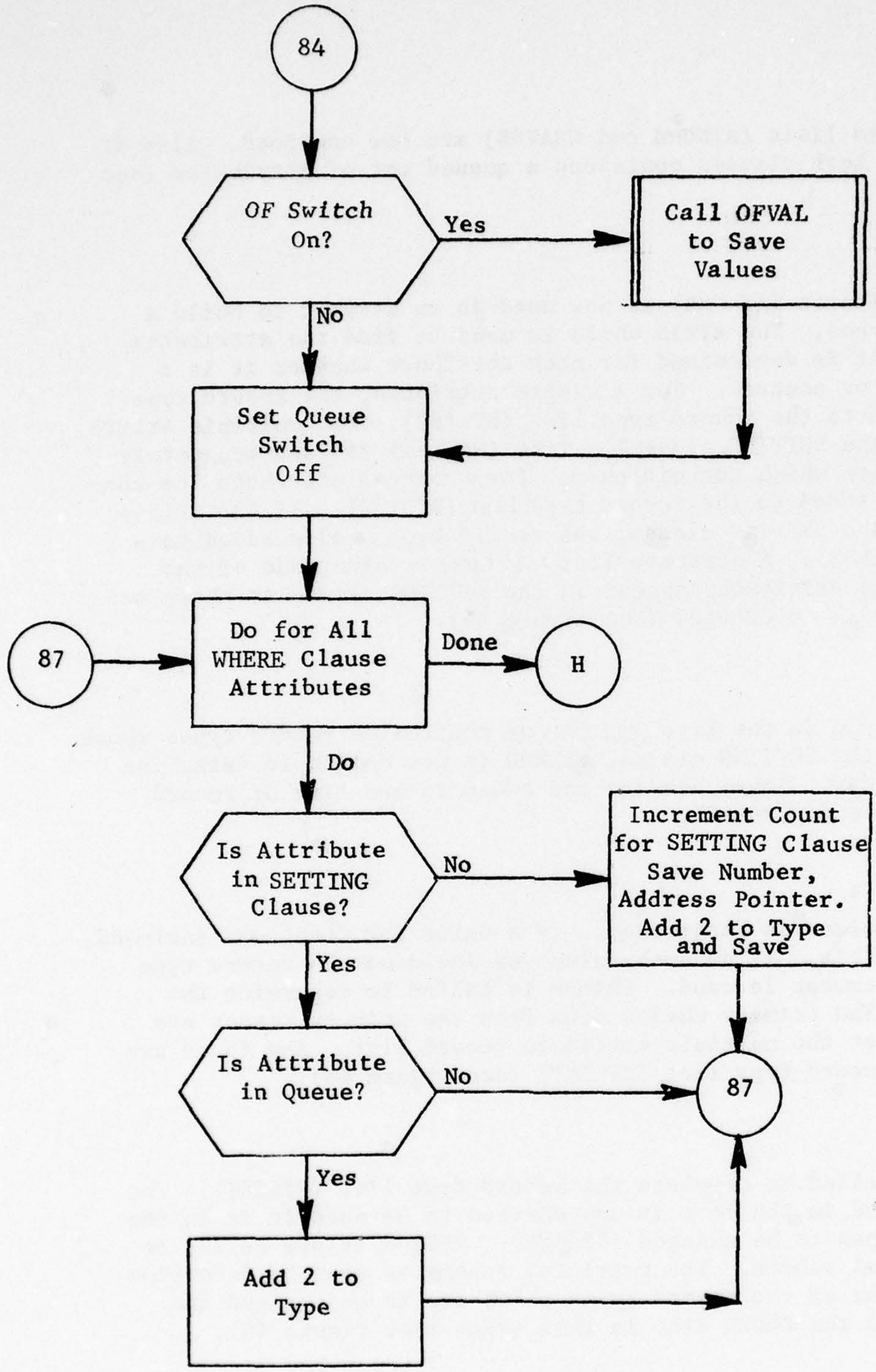



Figure 45. Subroutine CHANGE: Step Three
(Part 1 of 2)

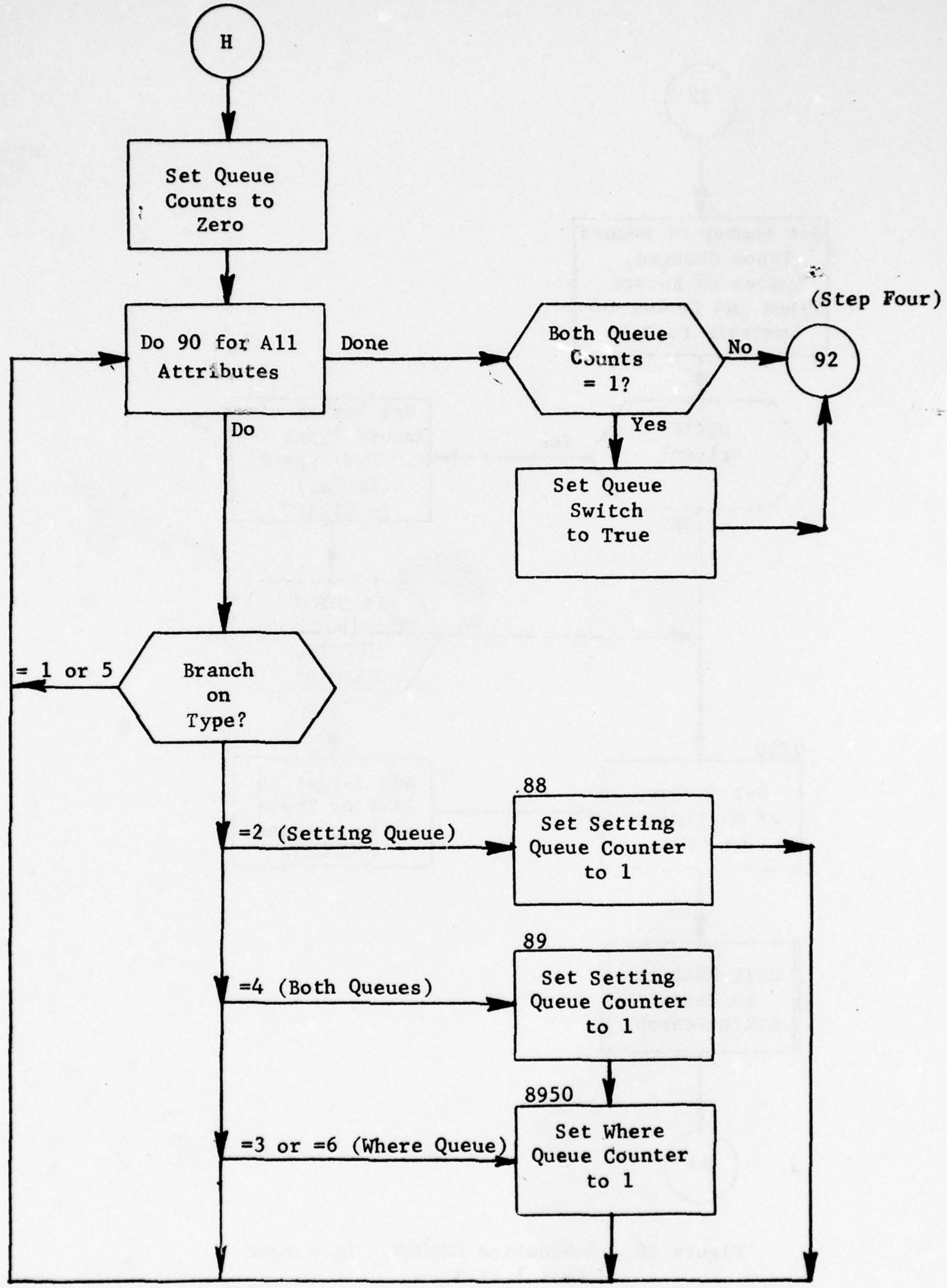



Figure 45. (Part 2 of 2)

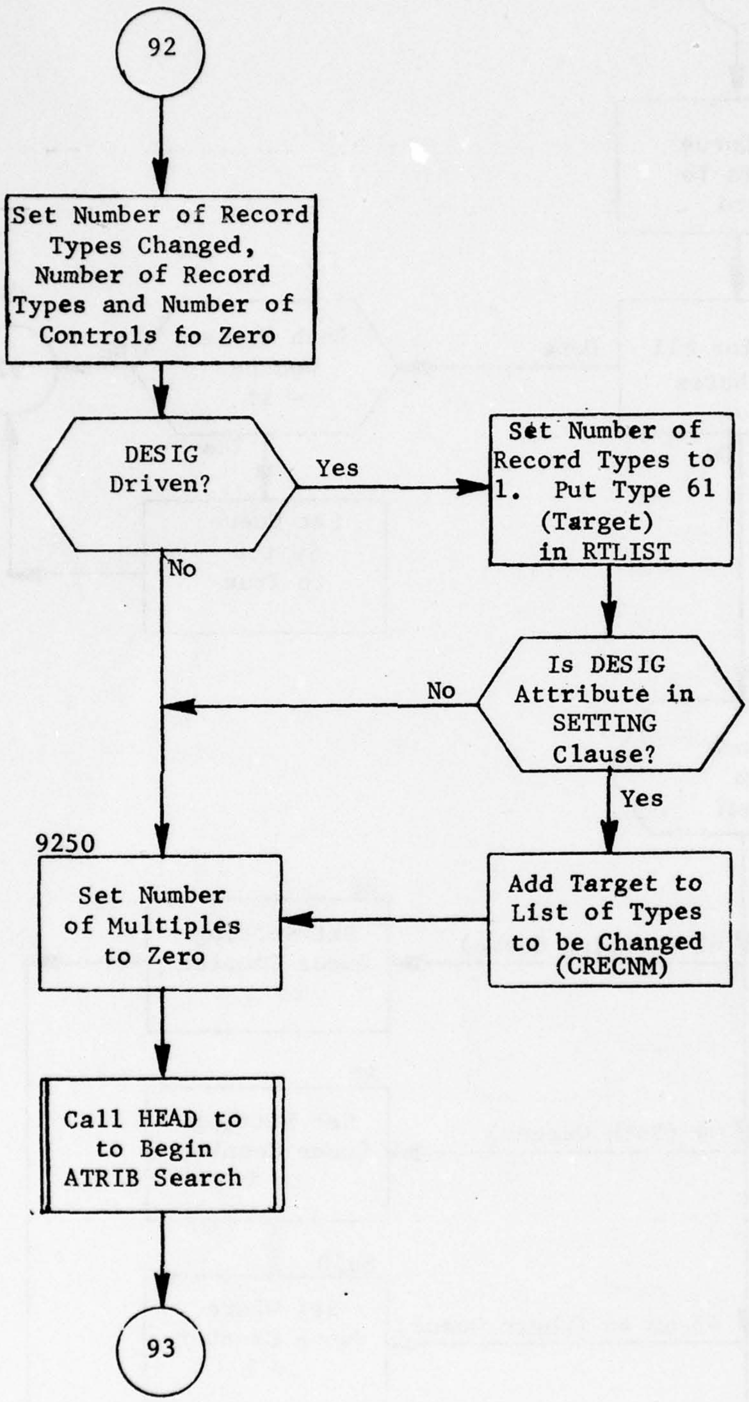



Figure 46. Subroutine CHANGE: Step Four
(Part 1 of 3)

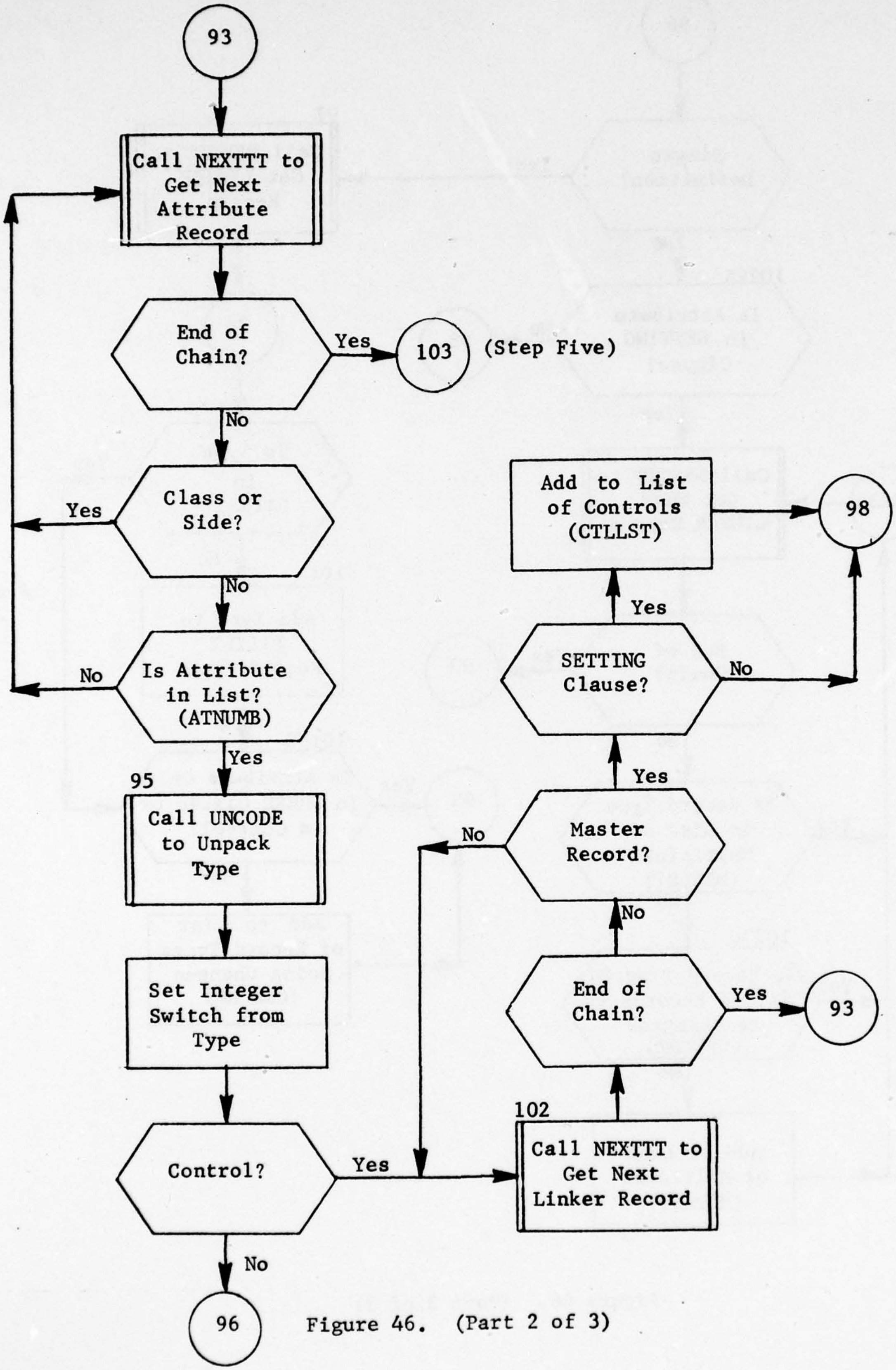



Figure 46. (Part 2 of 3)

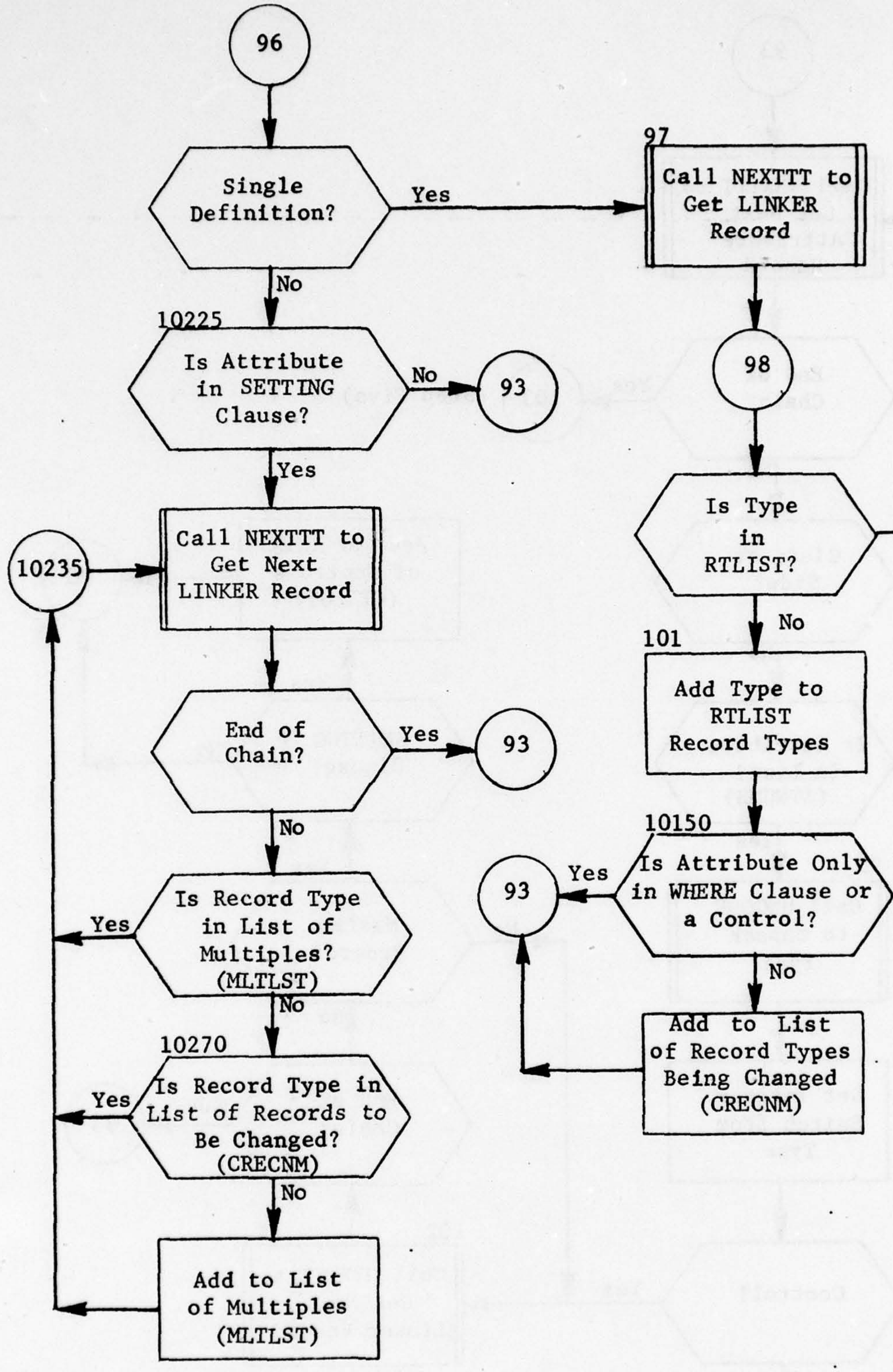



Figure 46. (Part 3 of 3)

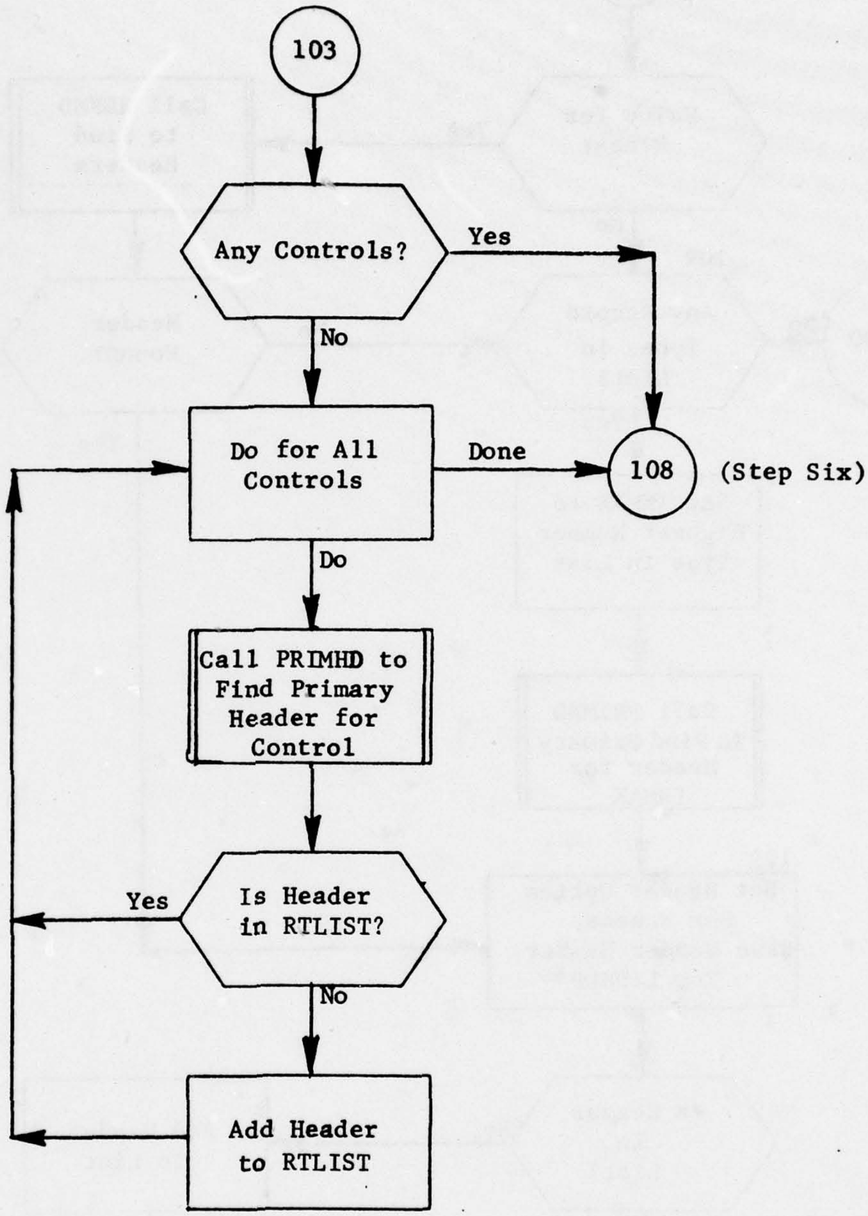



Figure 47. Subroutine CHANGE: Step Five

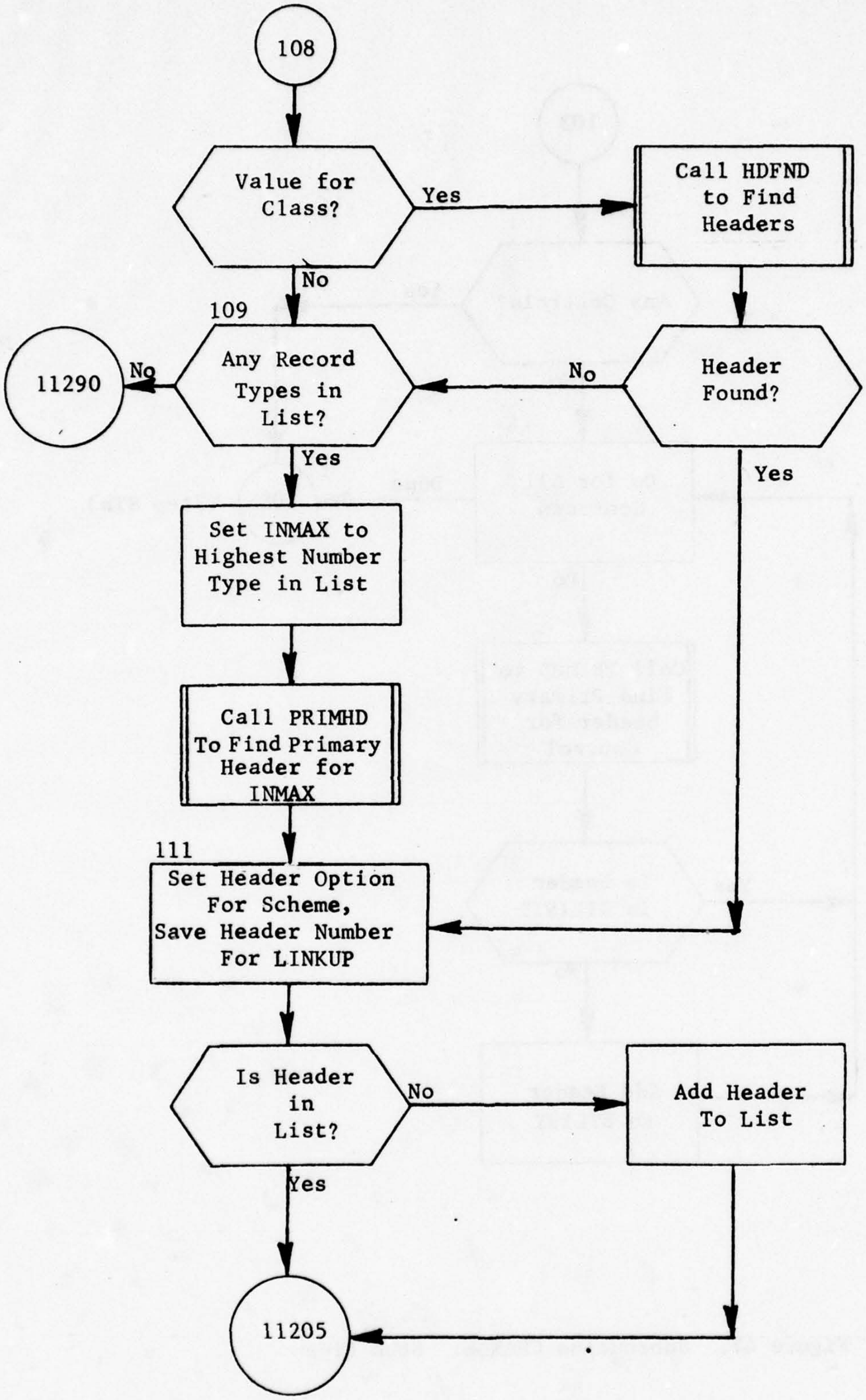



Figure 48. Subroutine CHANGE: Step Six
(Part 1 of 4)

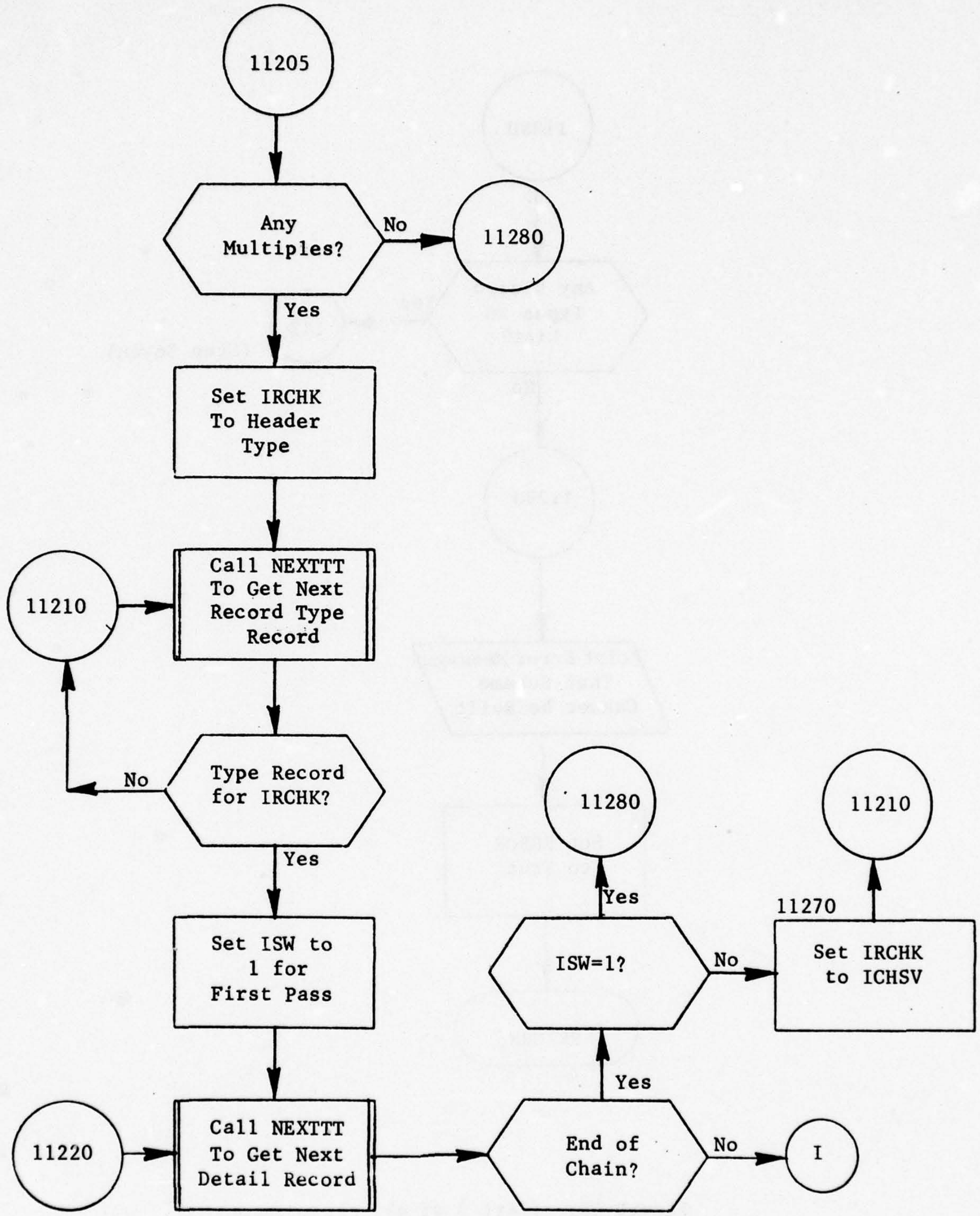



Figure 48. (Part 2 of 4)

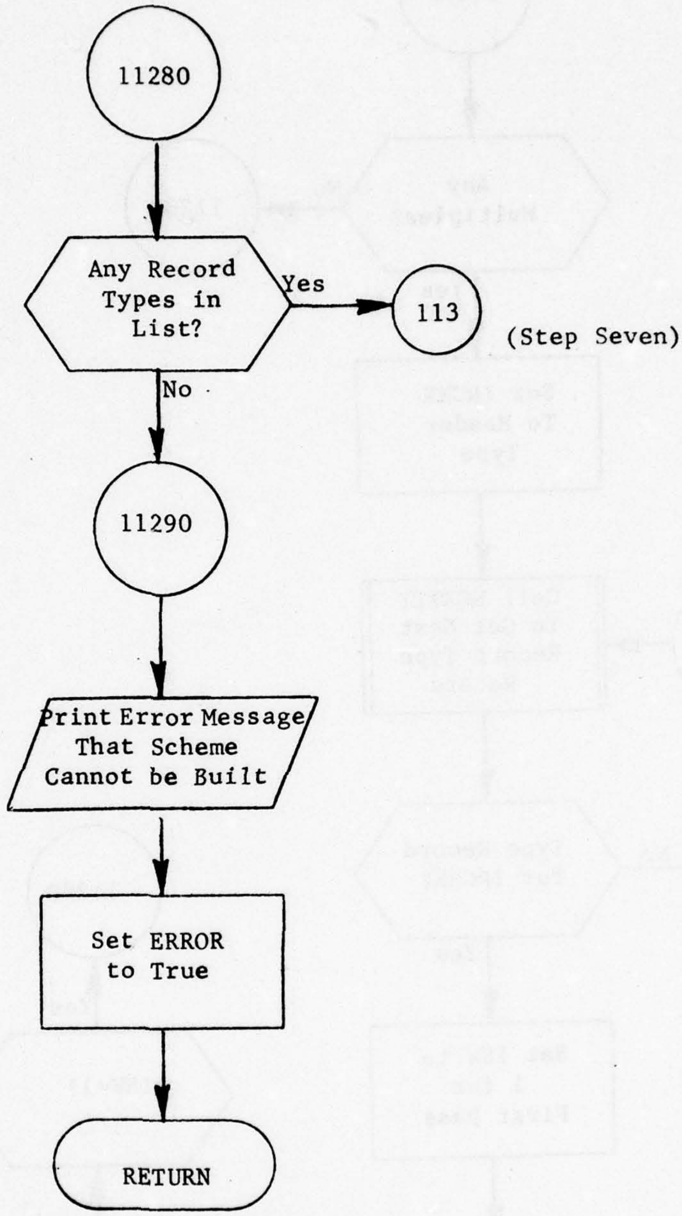



Figure 48. (Part 3 of 4)

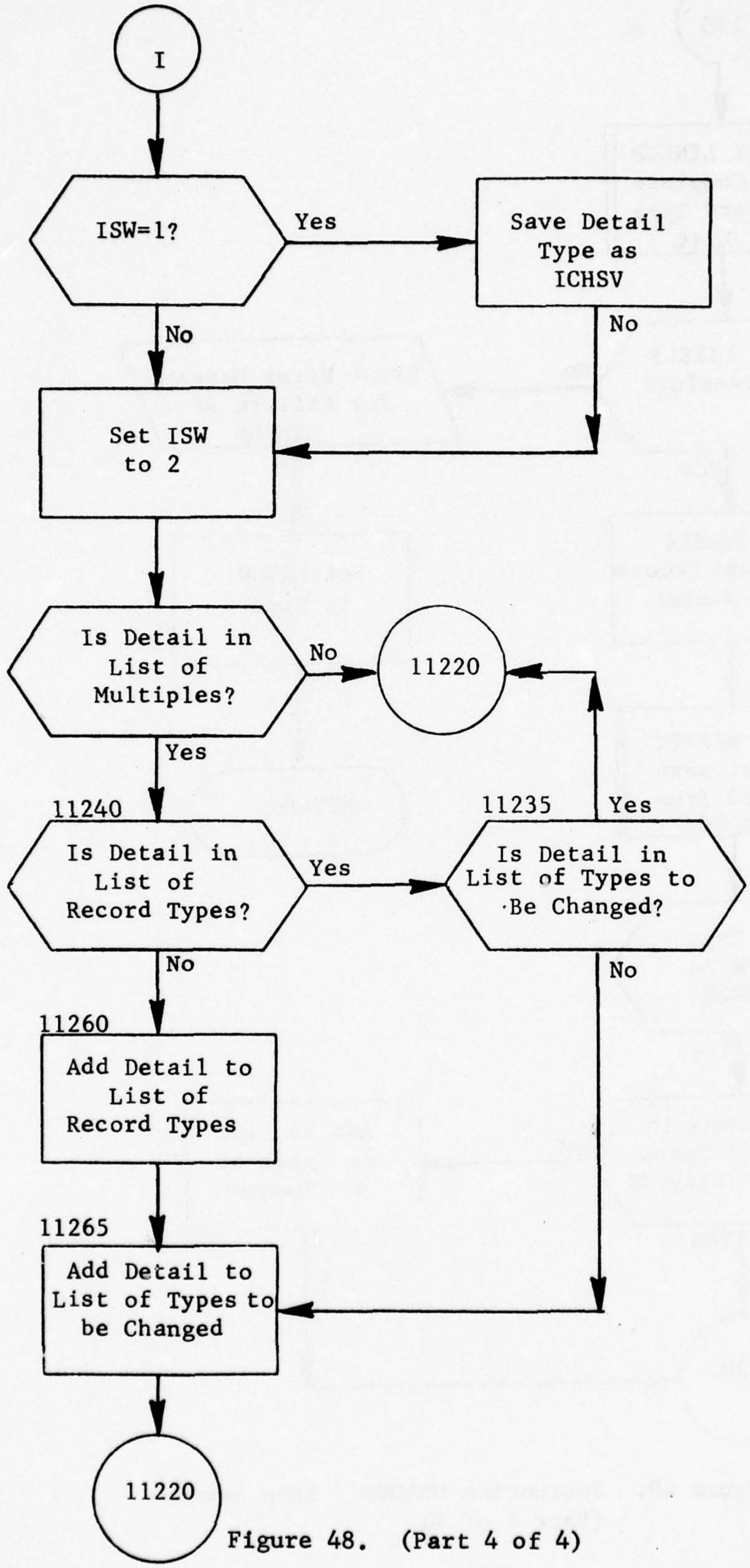


Figure 48. (Part 4 of 4)

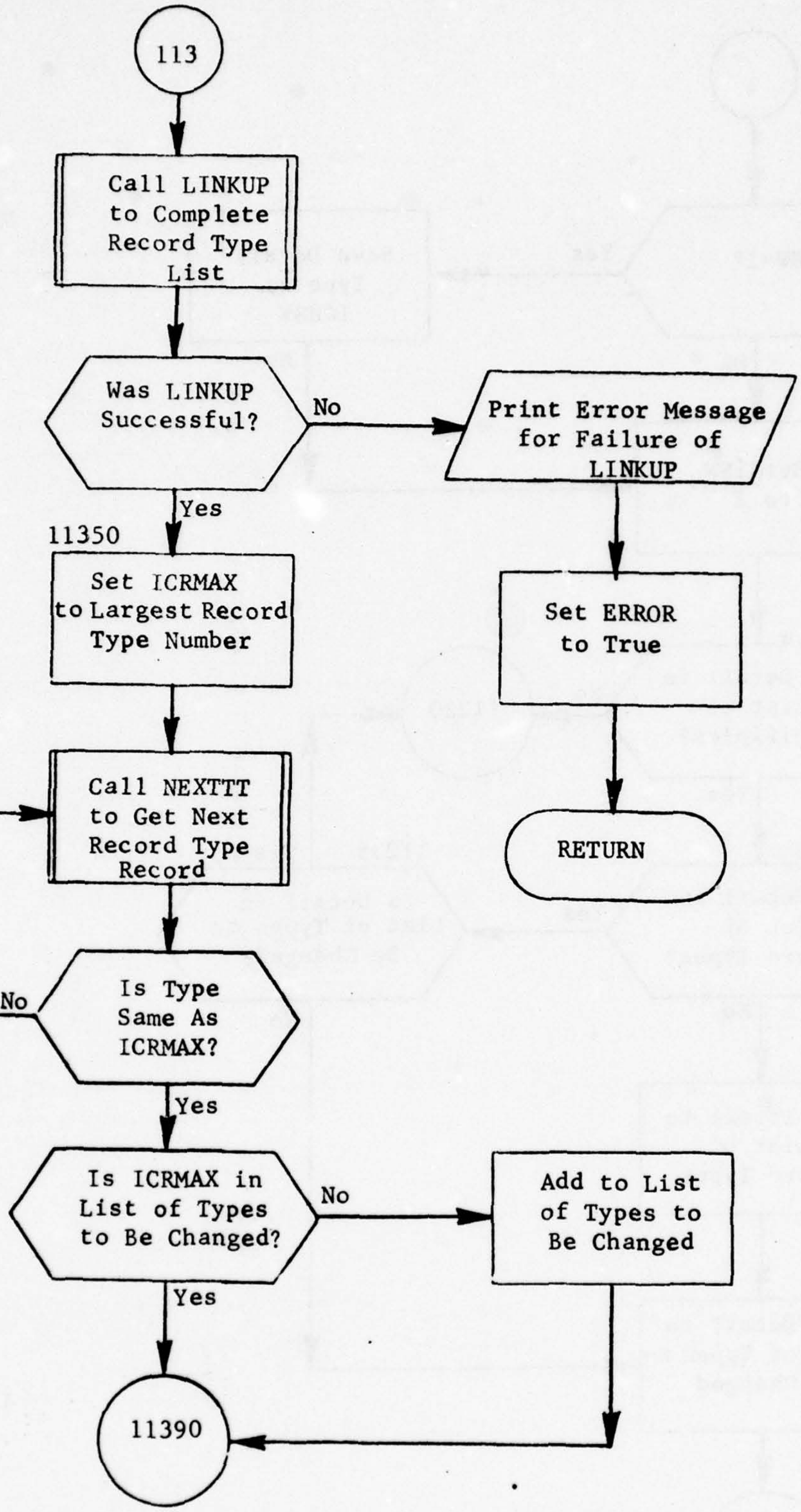



Figure 49. Subroutine CHANGE: Step Seven
(Part 1 of 5)

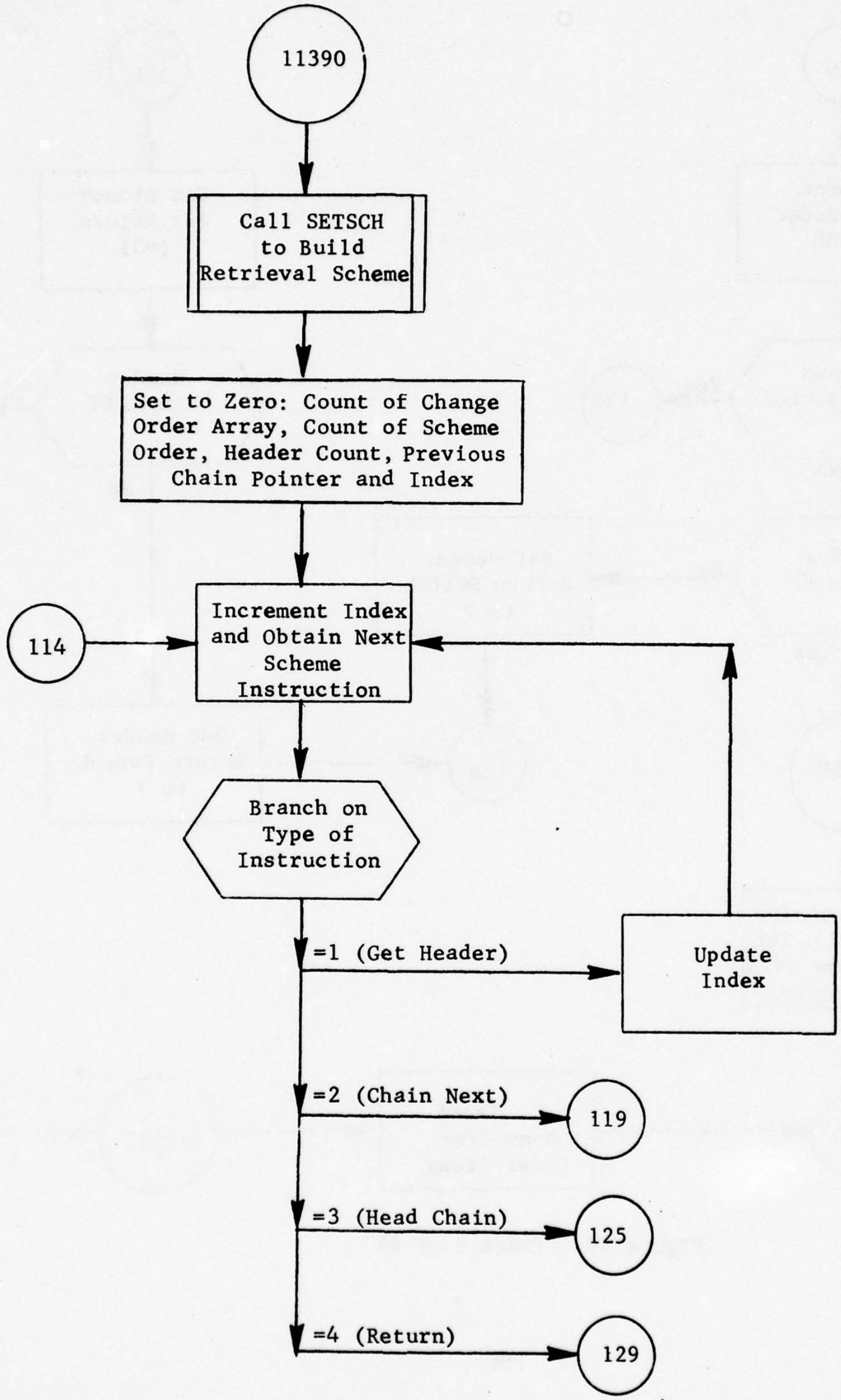



Figure 49. (Part 2 of 5)

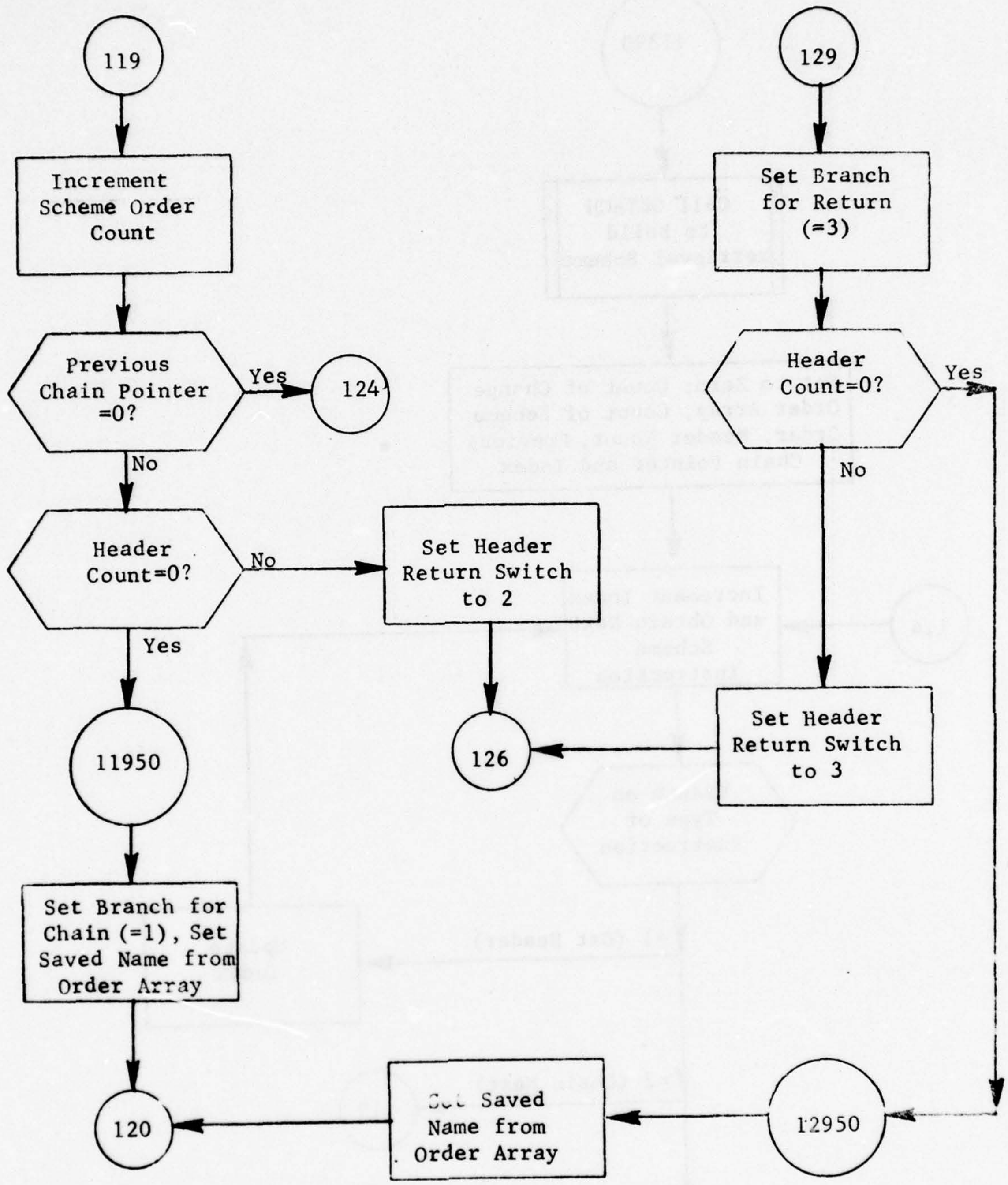



Figure 49. (Part 3 of 5)

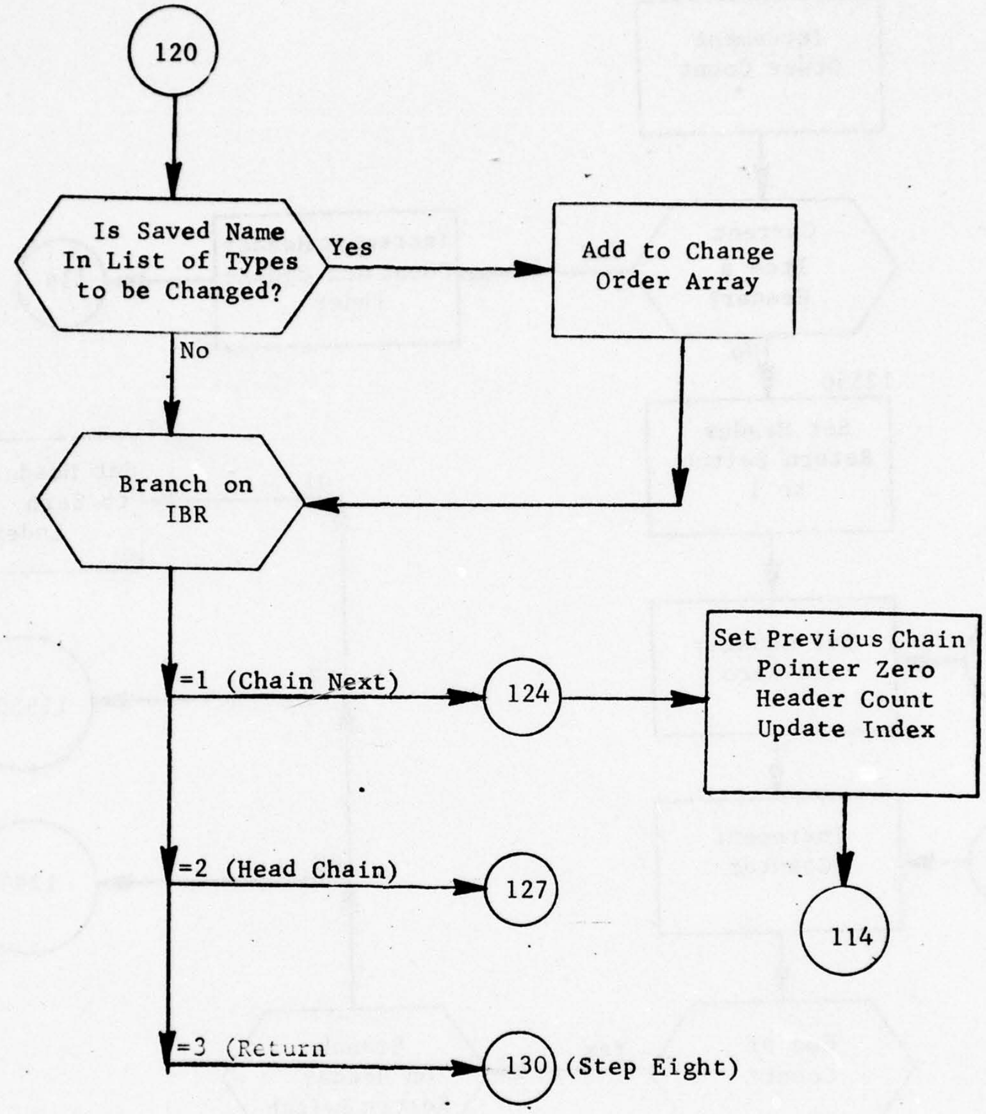



Figure 49. (Part 4 of 5)

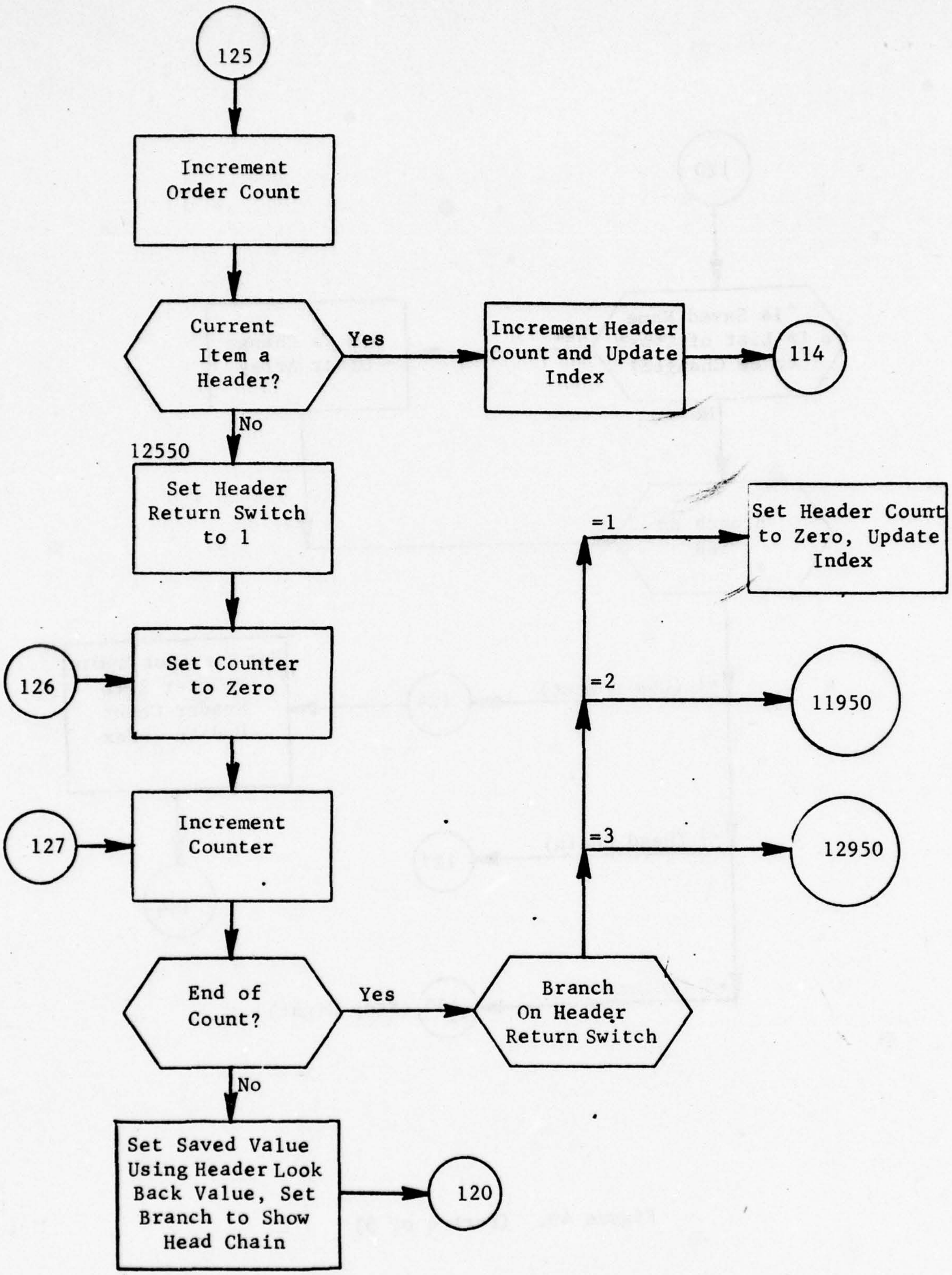



Figure 49. (Part 5 of 5)

### Step Eight

First, if the WHERE clause contained only the DESIG attribute, DESSCH is called to build a DESIG driven retrieval scheme (see section 4.8.1). Then the queue of DESIG values is processed one at a time. For each value NXTDES is called to retrieve the appropriate record. Then all SETTING clause attributes are set to their input values. The queued attributes are set to the values whose position in the queue correspond to the position of the DESIG value in its queue. Then the record types in the CHORD array have MODFY called for them in the order they appear.

If the retrieval is not DESIG driven the process is similar. The GETNXT routine is called until it passes back the indication that the retrieval process is over (ISW=2). For each record retrieved, XWHERE is called and if the record is one of those desired, the attribute values are set. If there is a queue, the WHERE is searched for the appropriate match and the corresponding SETTING queue values are used. Then MODFY is called for the elements of the CHORD array (see figure 50).

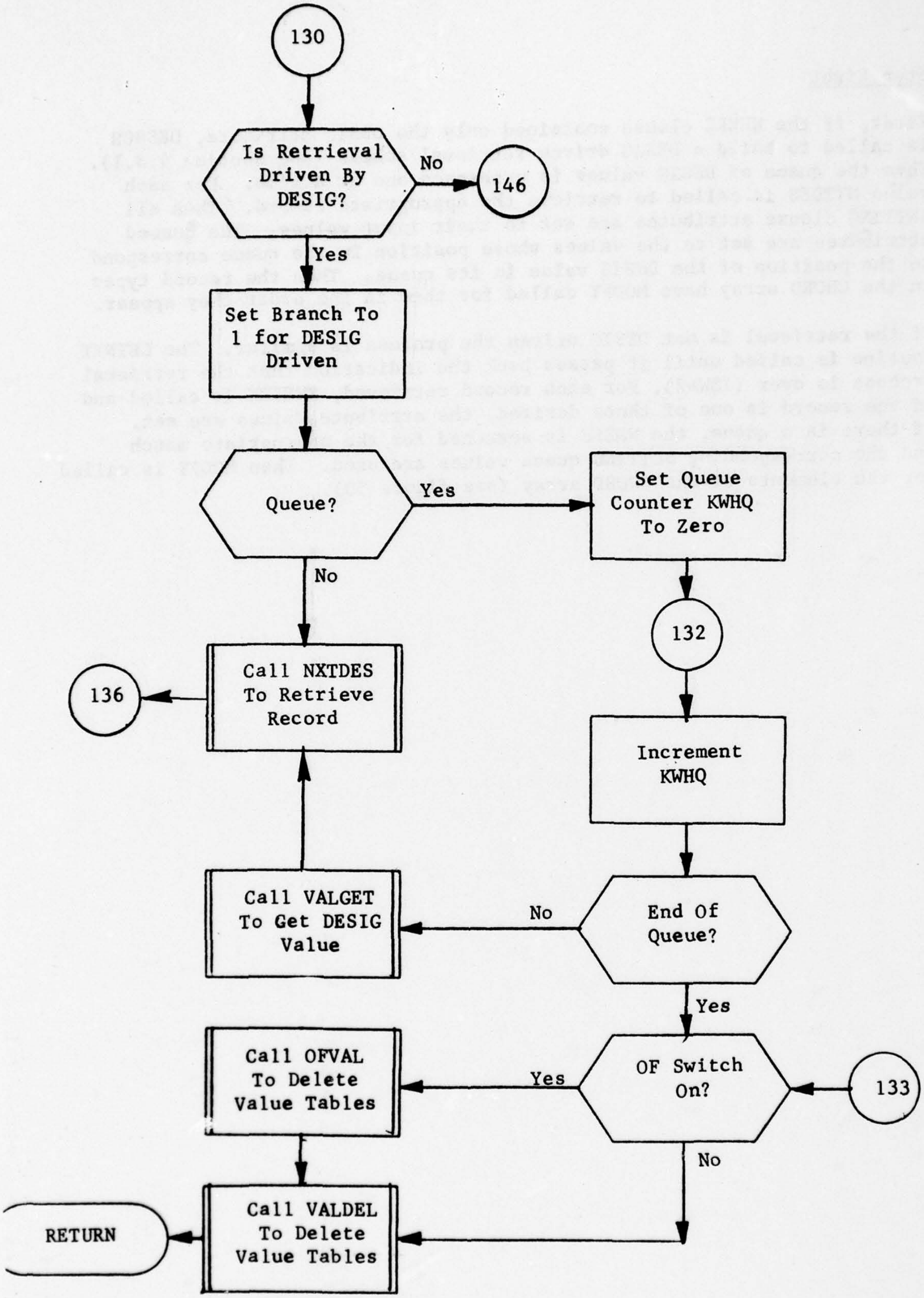



Figure 50. Subroutine CHANGE: Step Eight (Part 1 of 7)

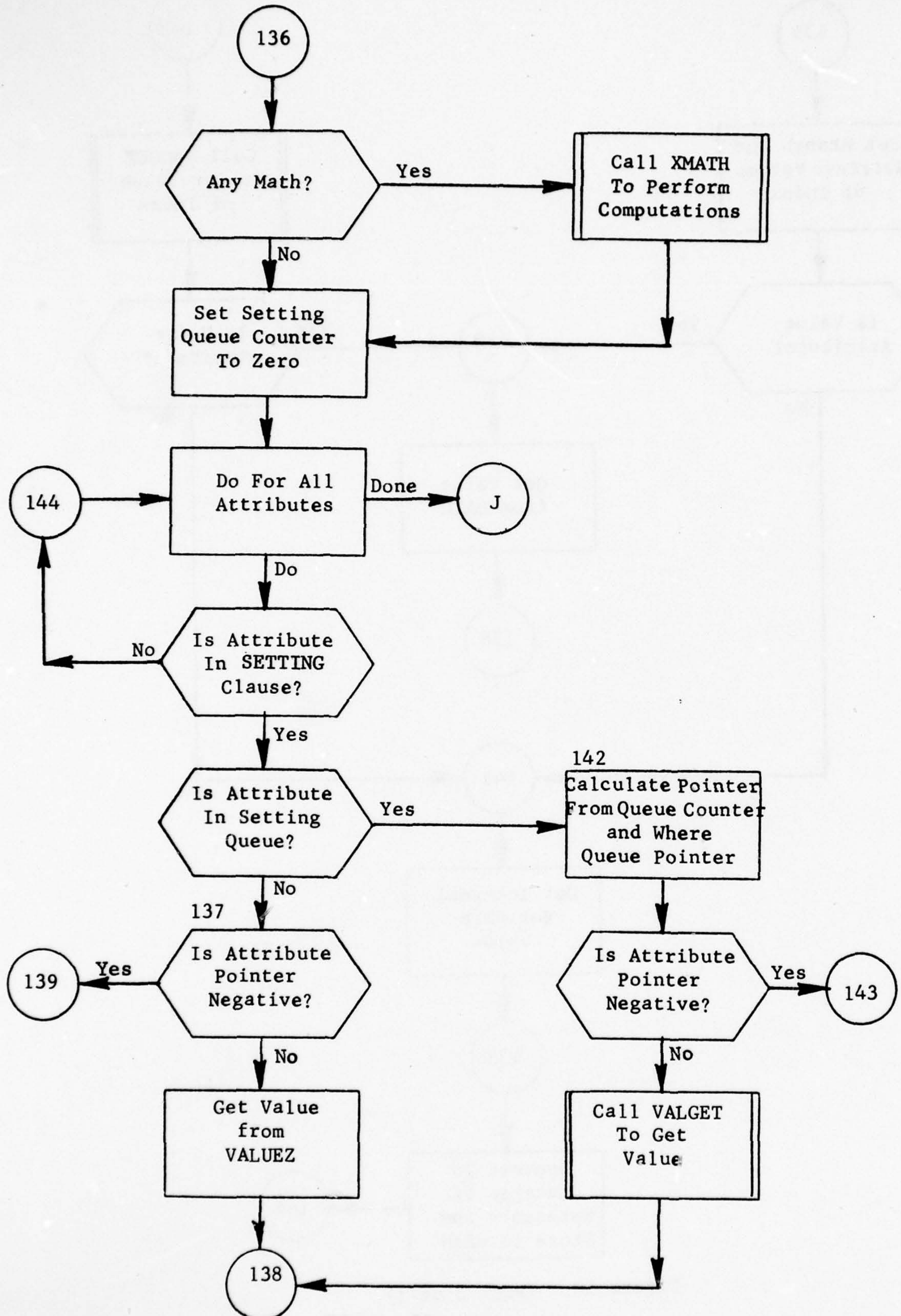



Figure 50. (Part 2 of 7)

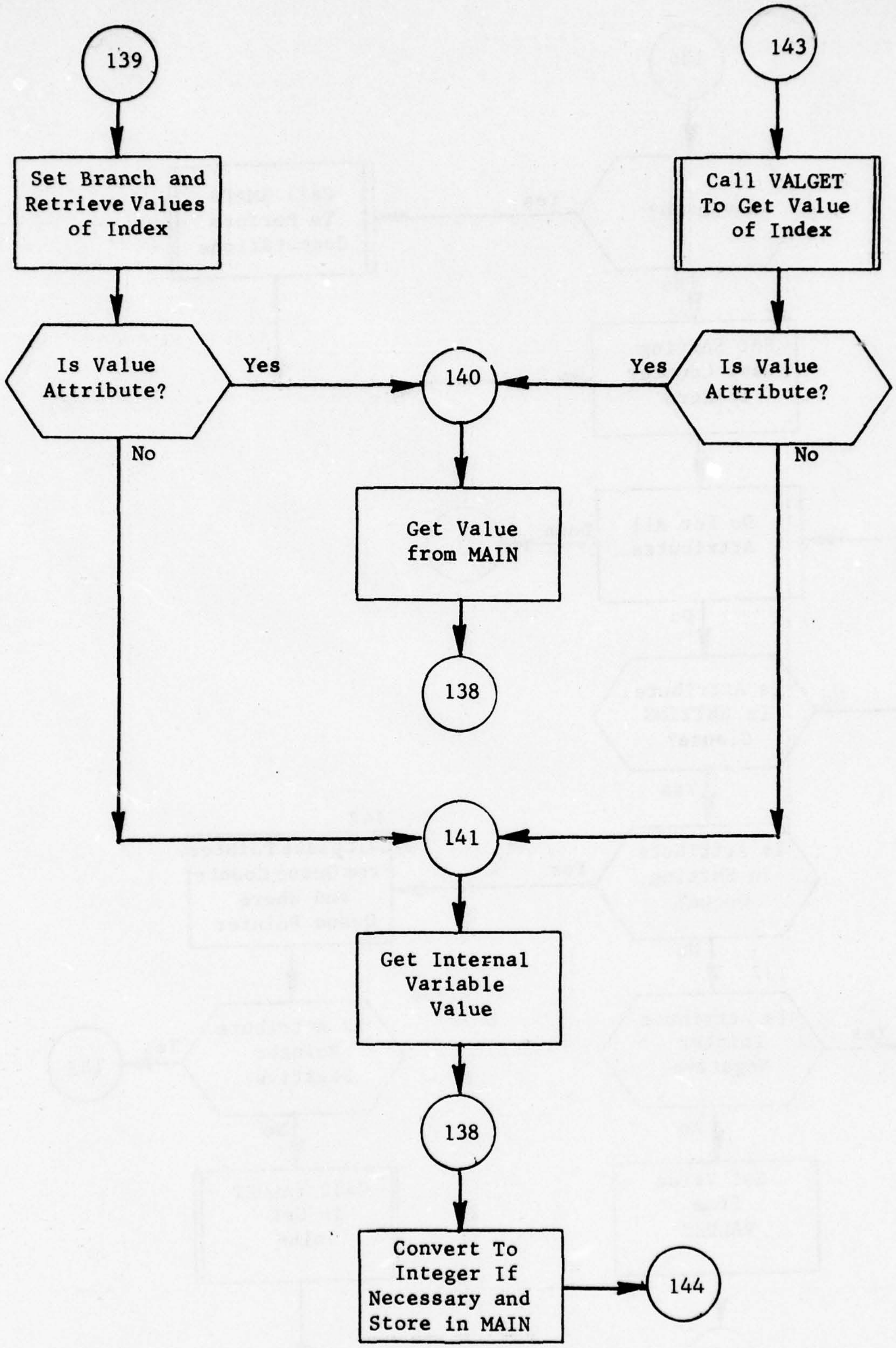



Figure 50. (Part 3 of 7)

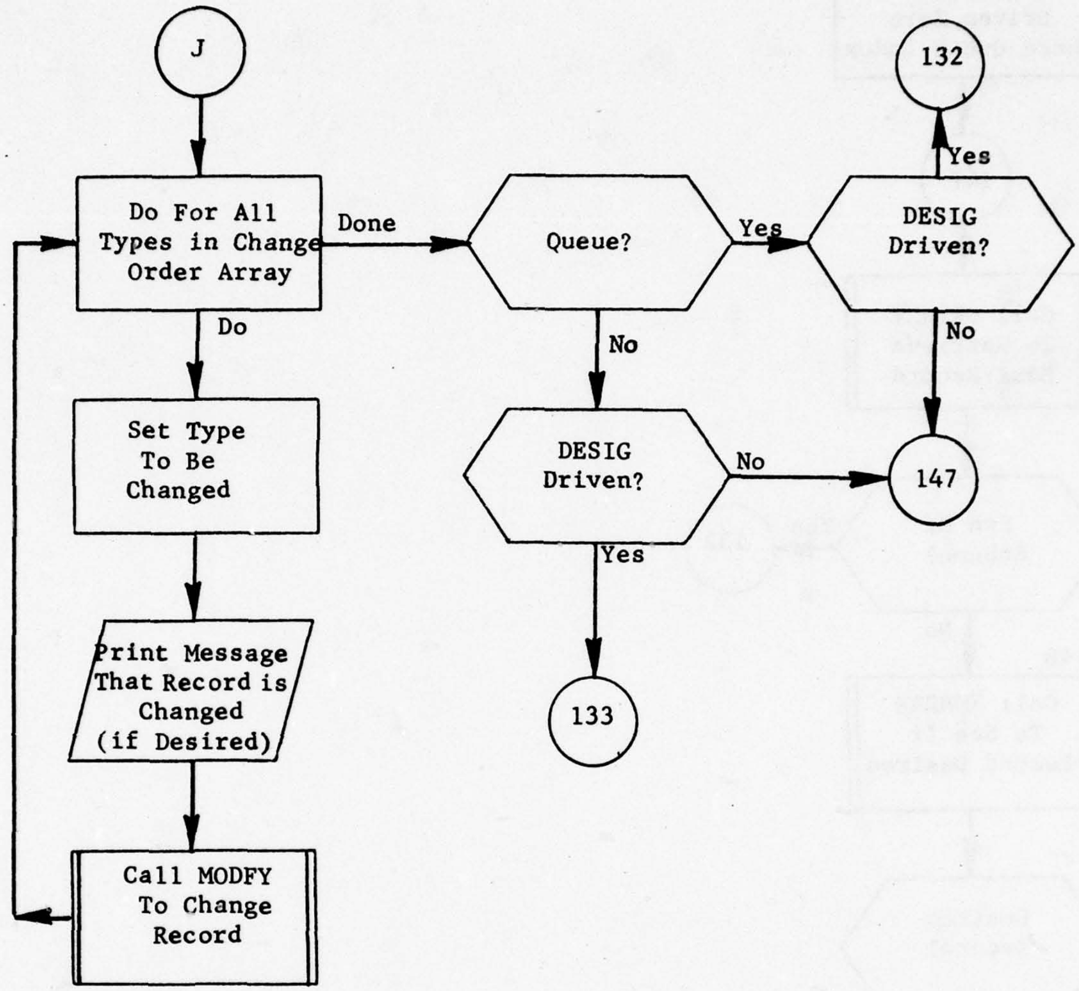



Figure 50. (Part 4 of 7)

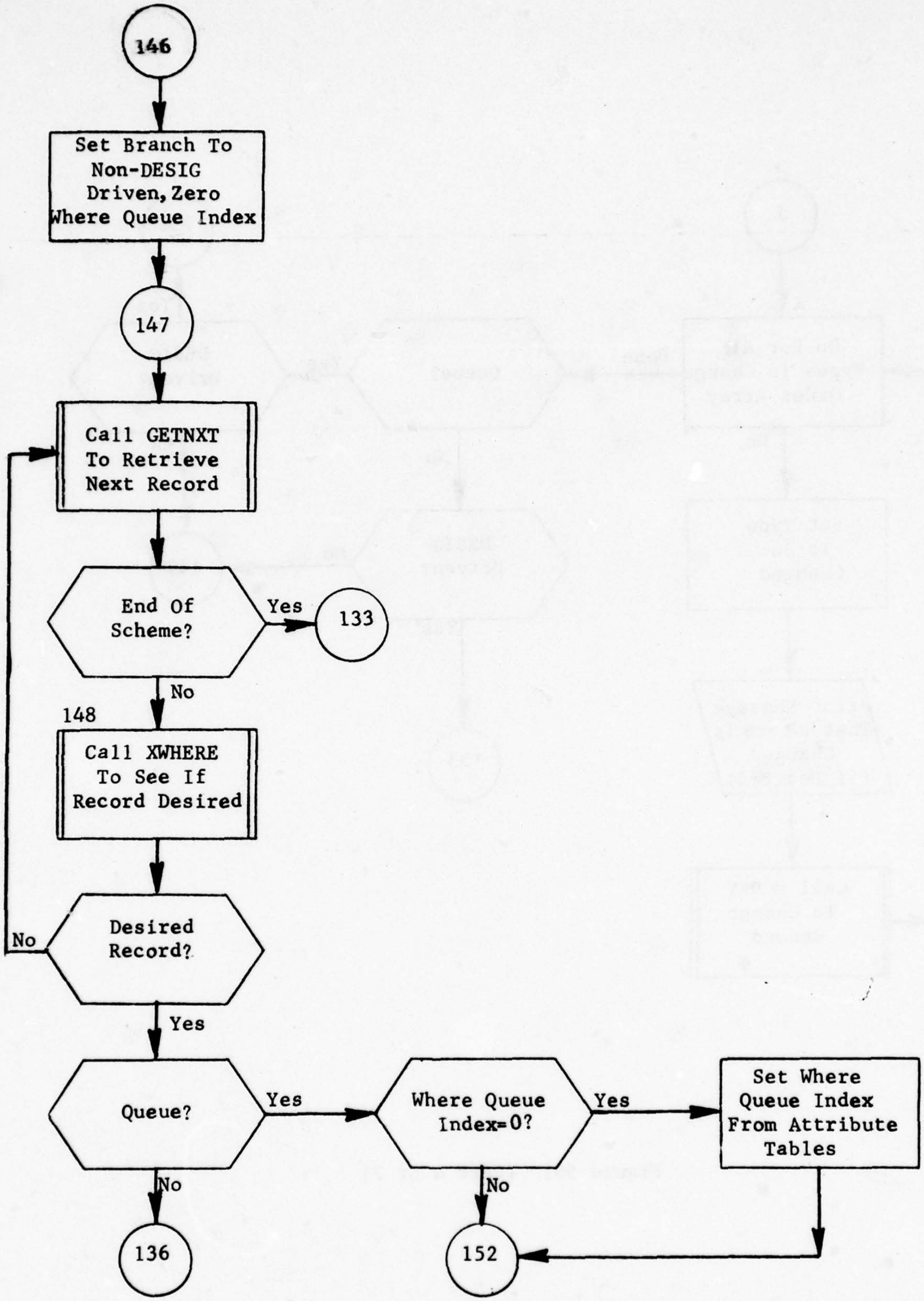



Figure 50. (Part 5 of 7)

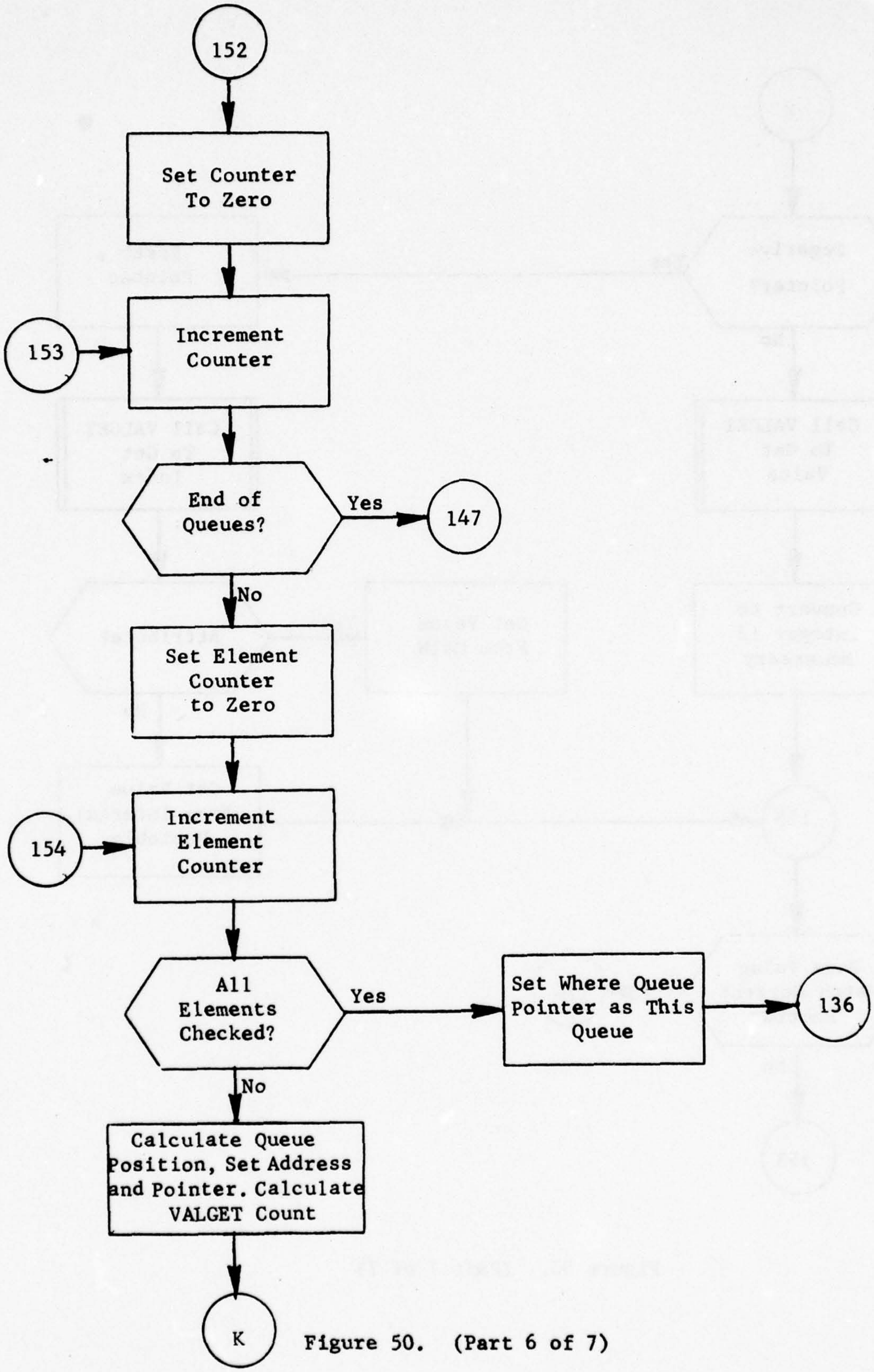



Figure 50. (Part 6 of 7)

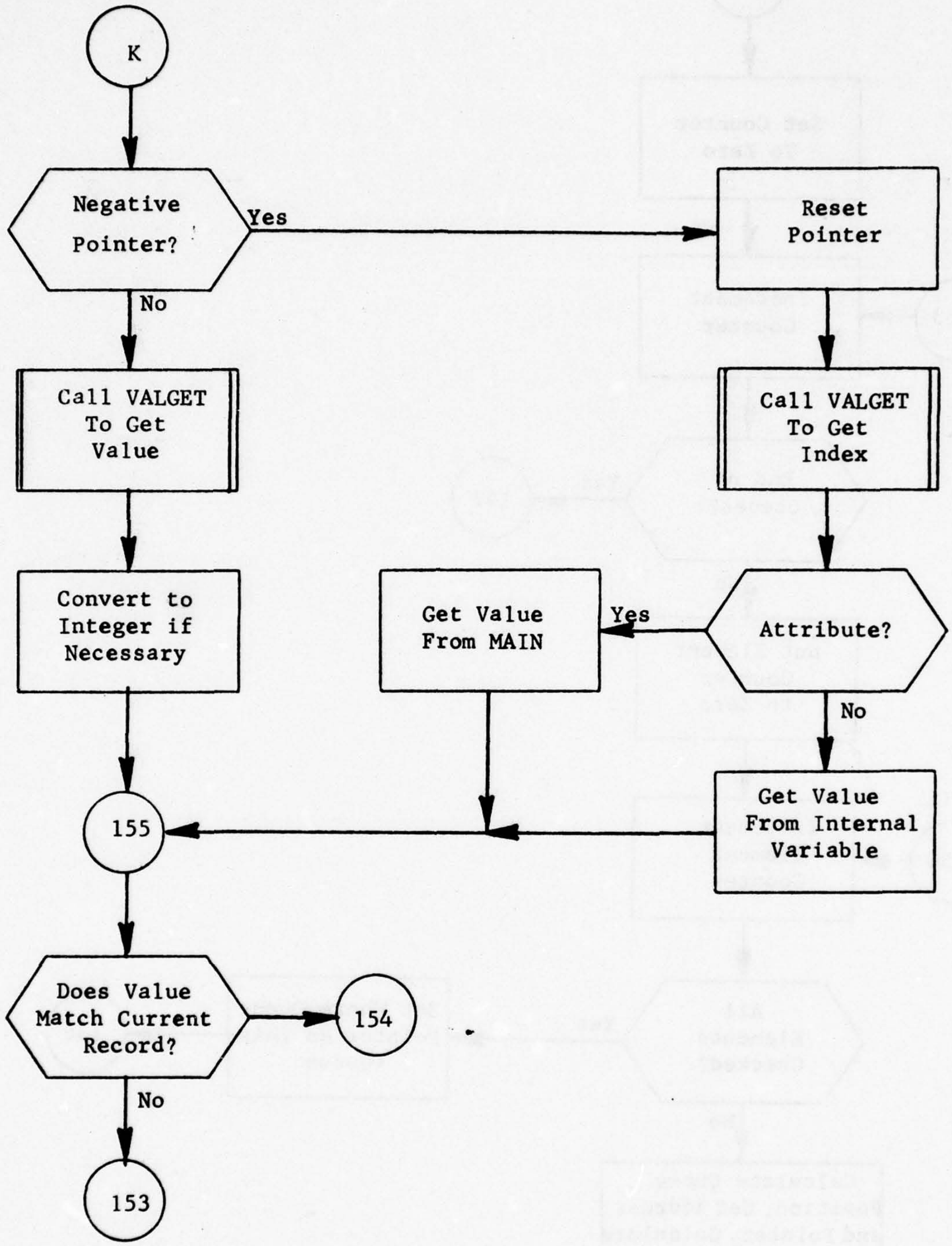



Figure 50. (Part 7 of 7)

### 4.8.1 Subroutine DESSCH

| | |
|---|---|
| PURPOSE: | To build a DESIG driven retrieval scheme |
| ENTRY POINTS: | DESSCH |
| FORMAL PARAMETERS: | LSTLST: Input list of record type numbers<br>LSTLN: Number of types in list |
| COMMON BLOCKS: | C10, C20, C30, OOPS, PRINSP, SCHEME, SCRTCH |
| SUBROUTINES CALLED: | NEXTTT |
| CALLED BY: | CHANGE |

Method:

This subroutine builds a special retrieval scheme driven by the attribute DESIG. The format of the scheme is a series of word pairs as outlined in table 16. After the invariable first instruction is created, the process begins by searching chains of which the TARGET record is master. For each such search that results in finding a record type in the record type list (LSTLST), a Chain Next instruction is added. The record types identified are placed in an open ended list (MIST). For each record type in MIST, the chains of which it is detail are searched for matches in LSTLST. Each match causes a chain head command to be inserted in the scheme and a new type added to the list. When all types have been processed, the return instruction is added.

Subroutine DESSCH is illustrated in figure 51.

Table 16. DESIG Driver Retrieval Scheme

| WORD 1 OF THE INSTRUCTION PAIR | WORD 2 OF THE INSTRUCTION PAIR | DESCRIPTION |
|---|---|---|
| 1 | 1 | Instructs NXTDES to retrieve the target using the input value for DESIG |
| 2 | Chain Name | Instructs NXTDES to call NEXTTT for the chain names |
| 3 | Chain Name | Instructs NXTDES to call HEAD for the chain named |
| 4 | 1 | Instructs NXTDES to reset POINT to 1 and return |

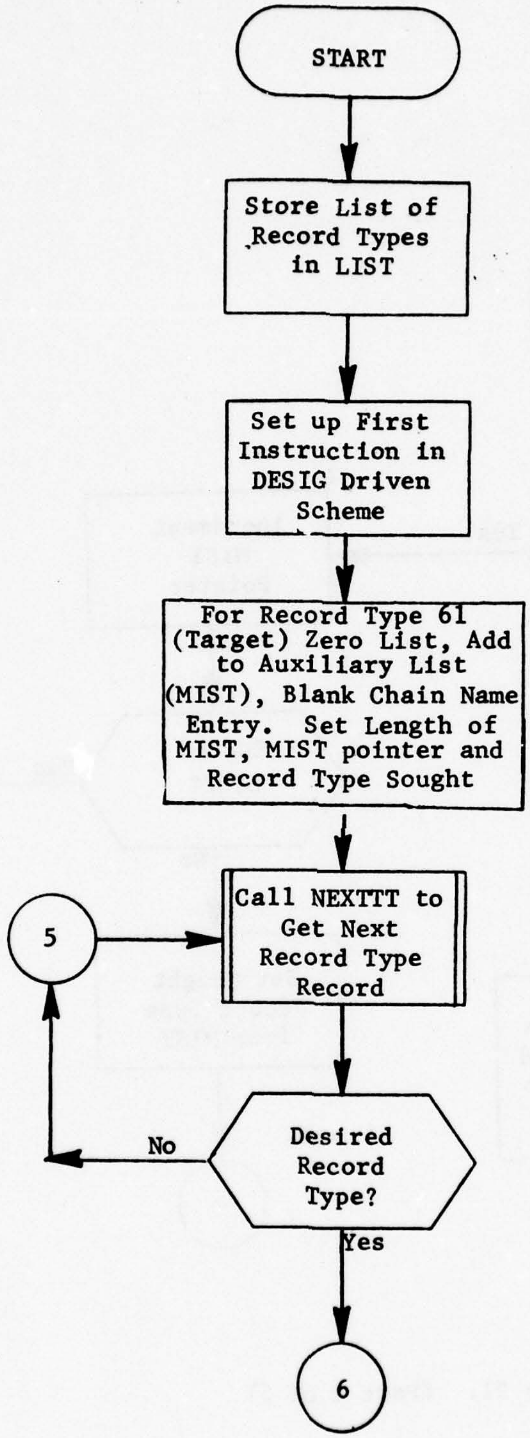



Figure 51. Subroutine DESSCH (Part 1 of 5)

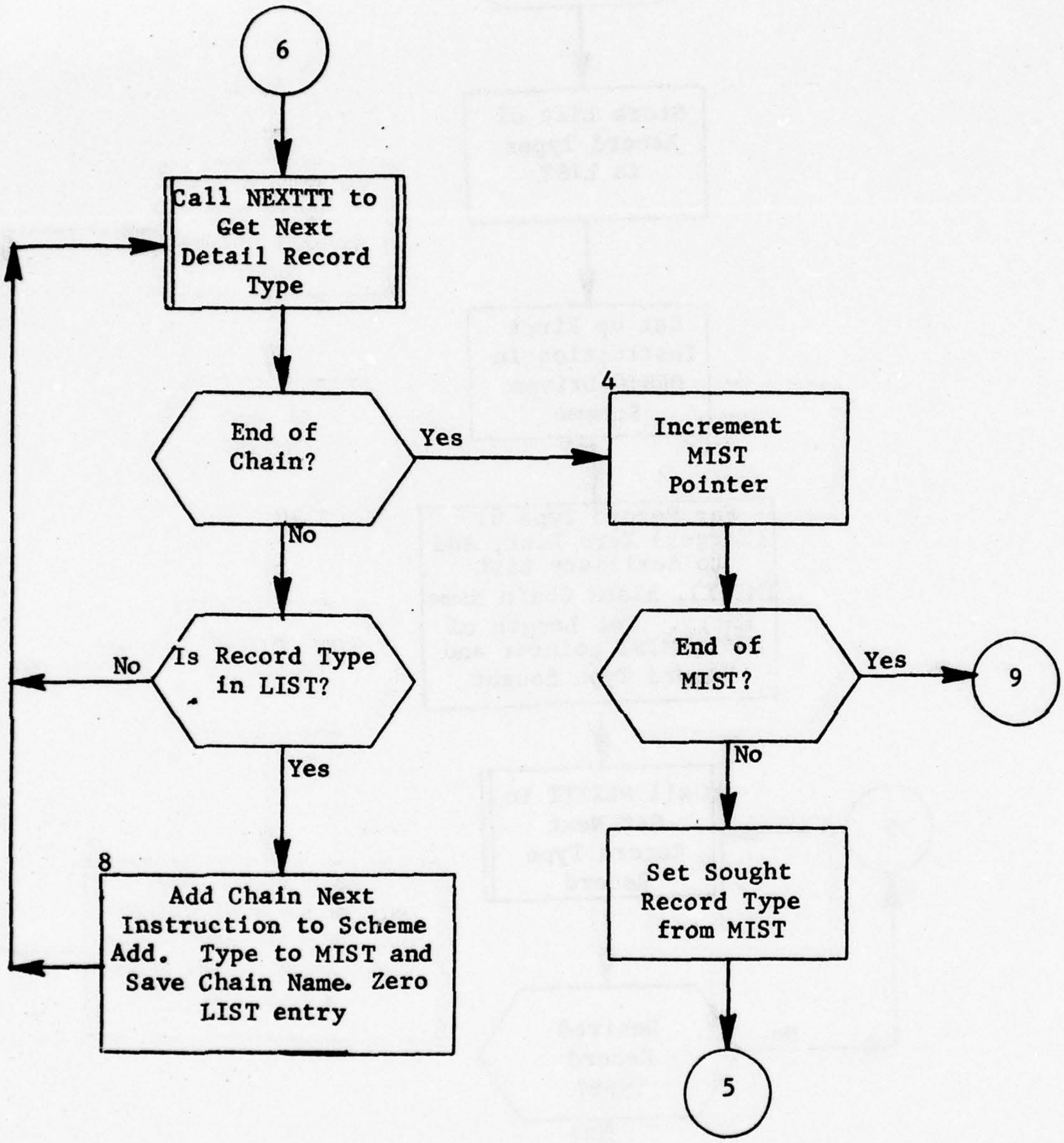



Figure 51. (Part 2 of 5)

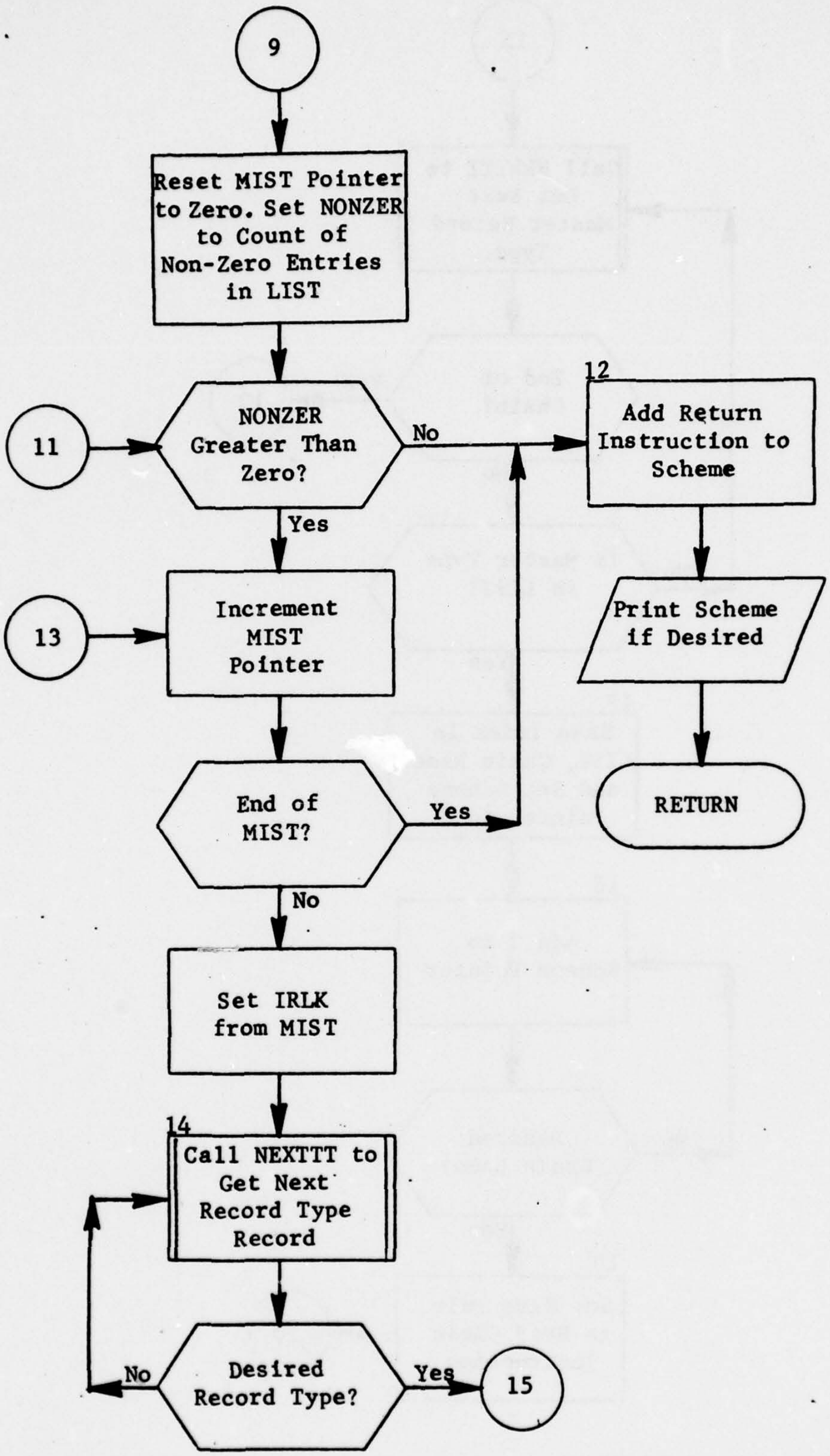



Figure 51. (Part 3 of 5)

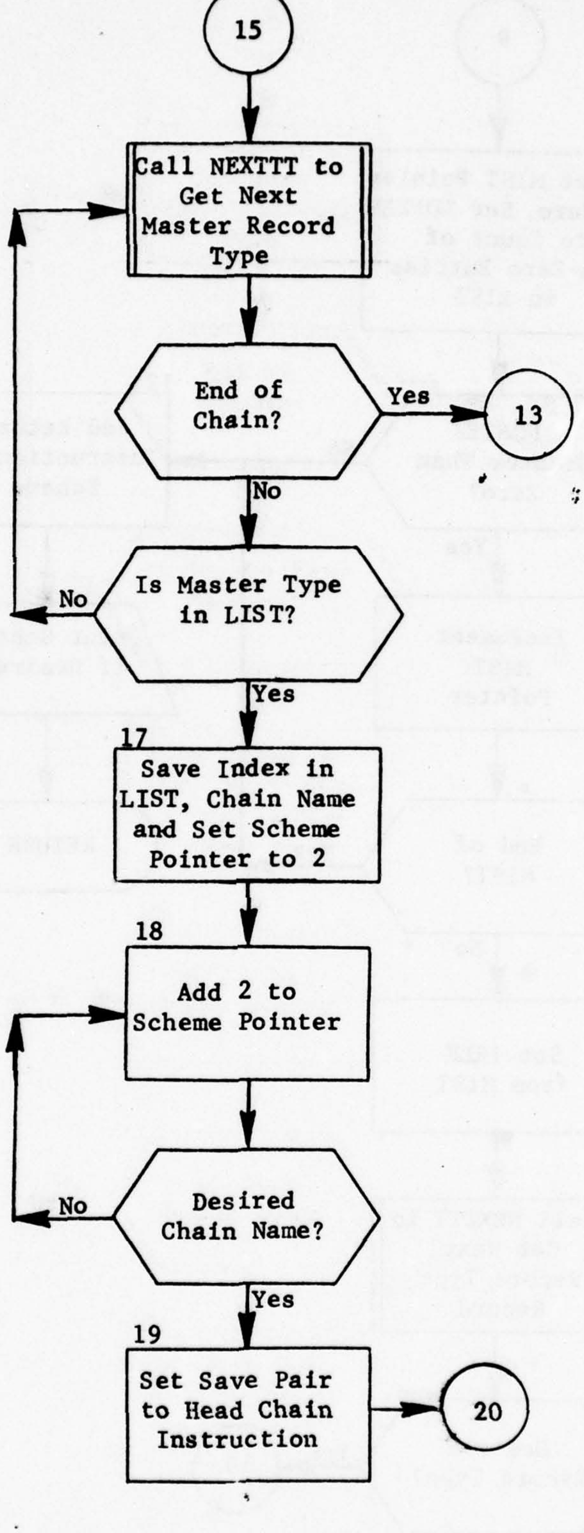



Figure 51. (Part 4 of 5)

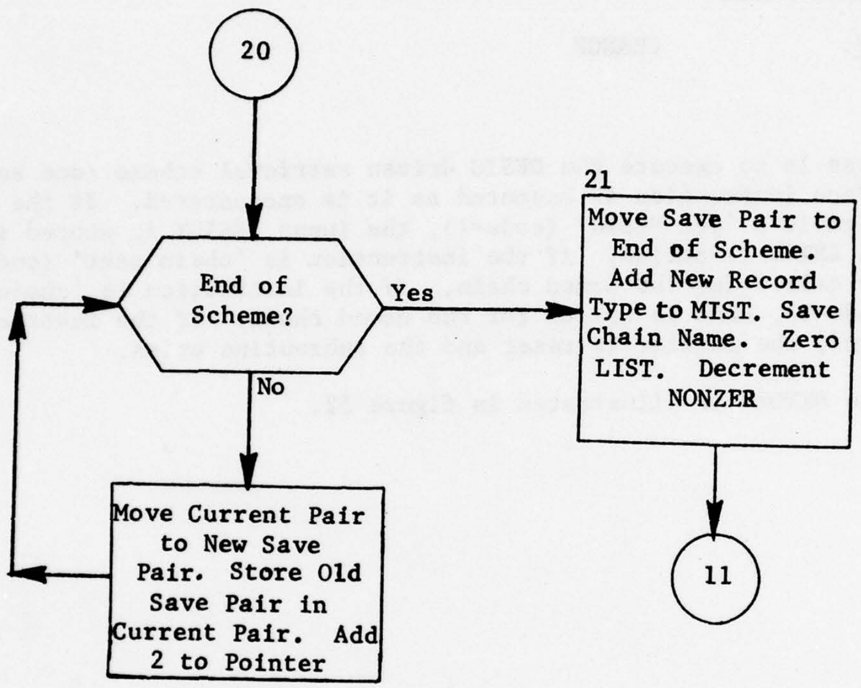



Figure 51. (Part 5 of 5)

### 4.8.2 Subroutine NXTDES

PURPOSE: To execute a DESIG driven retrieval scheme

ENTRY POINTS: NXTDES

FORMAL PARAMETERS: DESIGX: Input DESIG

COMMON BLOCKS: C30, SCHEME

SUBROUTINES CALLED: HEAD, NEXTTT, RETRV

CALLED BY: CHANGE

Method:

The process is to execute the DESIG driven retrieval scheme (see section 4.8.1). Each instruction is executed as it is encountered. If the instruction is a 'get DESIG' (code=1), the input DESIGX is stored in DESIG and RETRV is called. If the instruction is 'chain next' (code=2) NEXTTT is called for the named chain. If the instruction is 'chain head' (code=3), HEAD is called for the named chain. If the instruction is 'return', the pointer is reset and the subroutine exits.

Subroutine NXTDES is illustrated in figure 52.

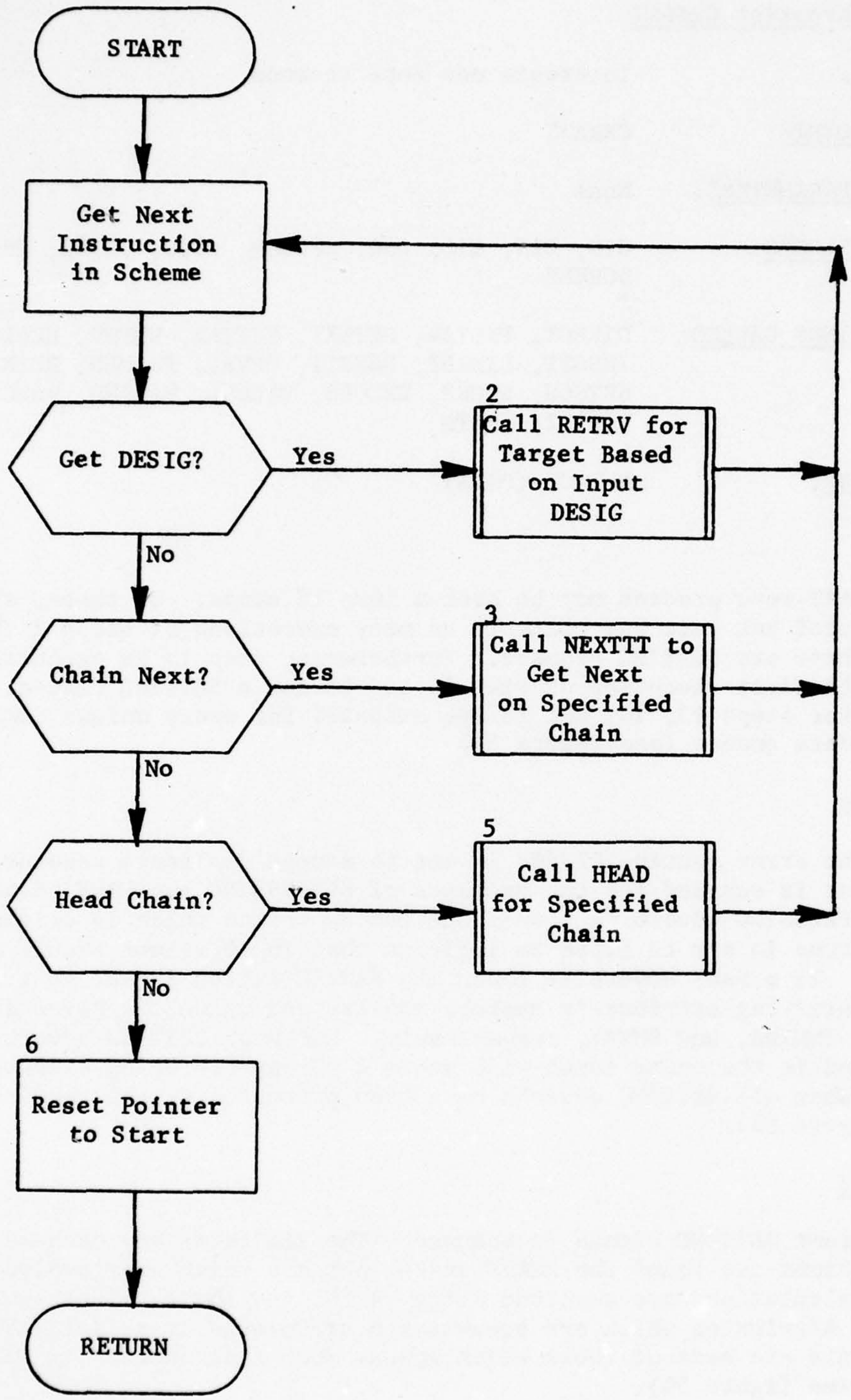



Figure 52. Subroutine NXTDES

## 4.9 Subroutine CREAAT

PURPOSE: To create new data records

ENTRY POINTS: CREAAT

FORMAL PARAMETERS: None

COMMON BLOCKS: C10, C15, C20, C30, ERRCOM, OOPS, ORDER, PRINSP, SCHEME

SUBROUTINES CALLED: DIRECT, FNDTAR, GETNXT, GETTAR, HDFND, HEAD, INSGET, LINKUP, NEXTTT, OFVAL, PRIMHD, RETRV, SETSCH, STORE, UNCODE, VALDEL, VALFND, VALGET, VALPUT, XMATH

CALLED BY: ENTMOD (DATA)

### Method:

The CREAAT verb process may be broken into 15 steps. Of these, step 1 is executed but once and controls as many executions of steps 2 through 14 as there are SETTING clauses. Furthermore, step 13 is executed after the first execution of step 14 and 15 for a SETTING clause. Thereafter steps 13, 14, and 15 are executed for every unique combination of any data queues (see figure 39)

### Step One

First the error routine ERPROC is set to accept duplicate records. Then the input is scanned for the presence of SUPRESSING and SAME adverbs. If a SUPRESSING adverb is found, the DRCTSW switch which is originally set to true is set to false to indicate that input values should not be edited. If a SAME adverb is found the SAMESW switch is set to true and the identifying attribute's number, address and value are saved in ISMIDN, ISMIDA, and SMVAL, respectively. Now each SETTING adverb is processed in the order input with steps 2 through 14 being executed for each. When all SETTING adverbs have been processed the subroutine exits (see figure 53).

### Step Two

The current SETTING clause is scanned. The limits of any mathematical calculations are found for XMATH and OF phrases which are involved in those calculations are resolved using VALFND and their values saved with OFVAL. Attributes which are encountered are placed in a list (ATNUMB) and counts are made of those which appear more than once or in collections (see figure 54).

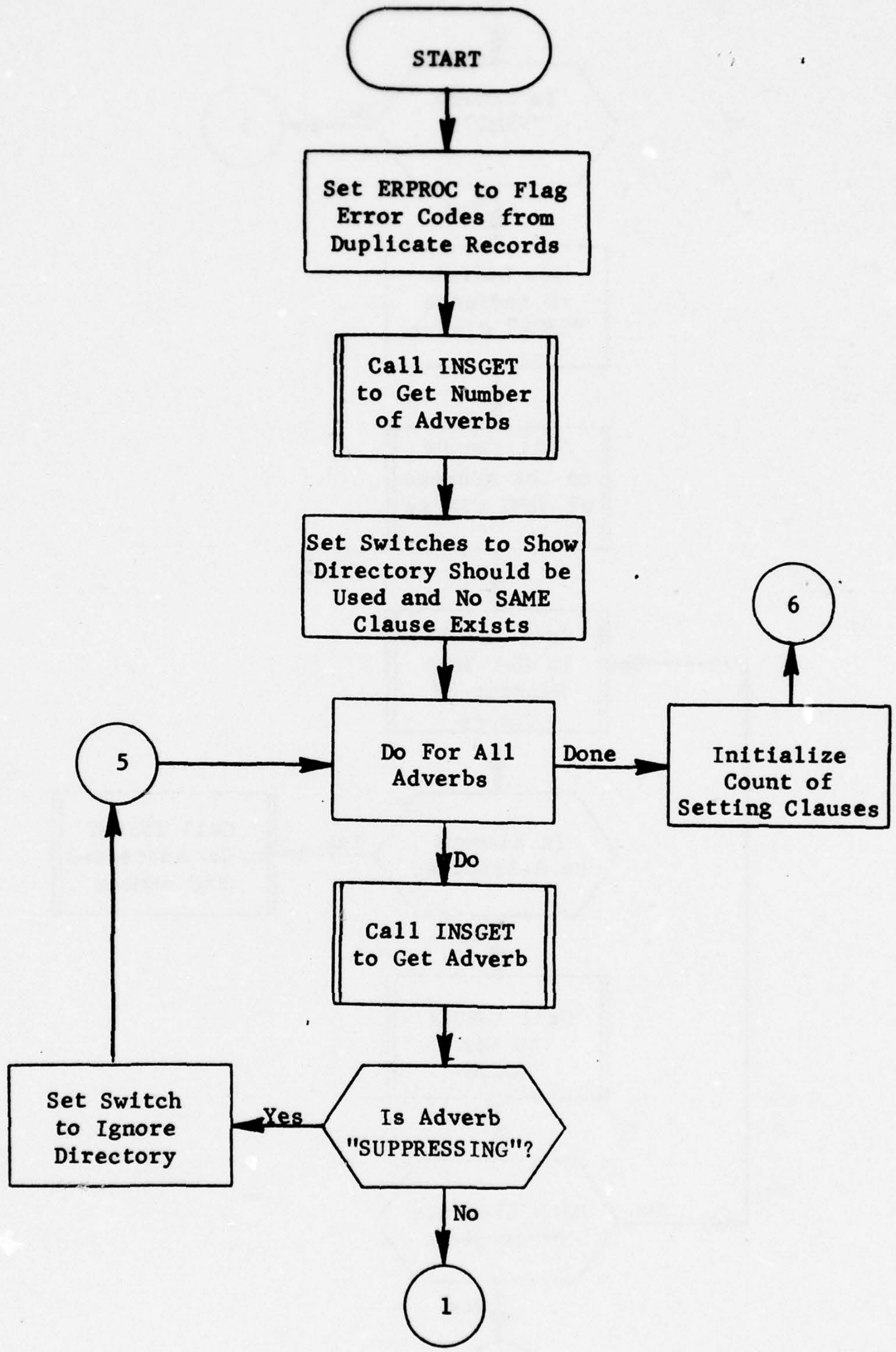



Figure 53. Subroutine CREAAT: Step 1 (Part 1 of 3)

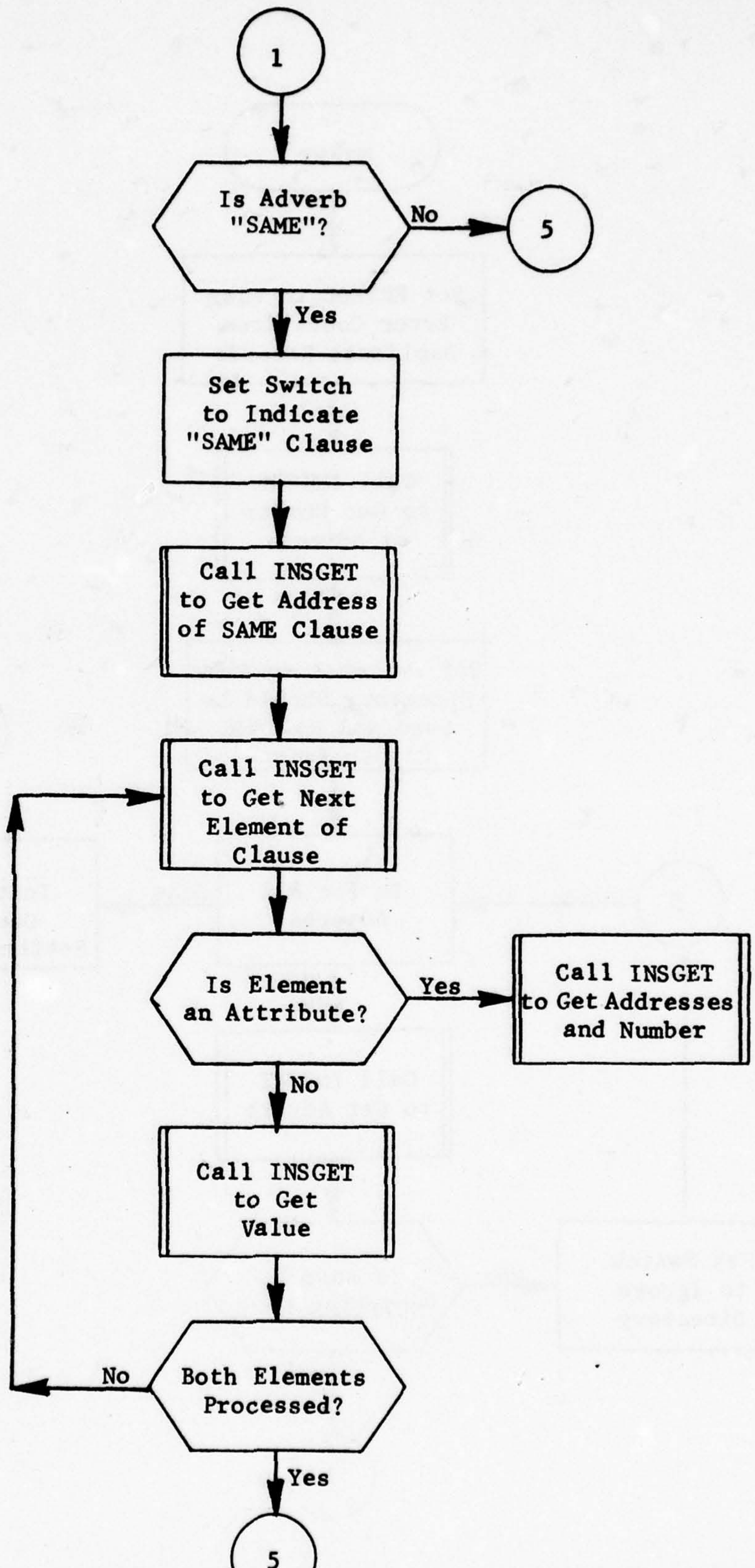



Figure 53. (Part 2 of 3)

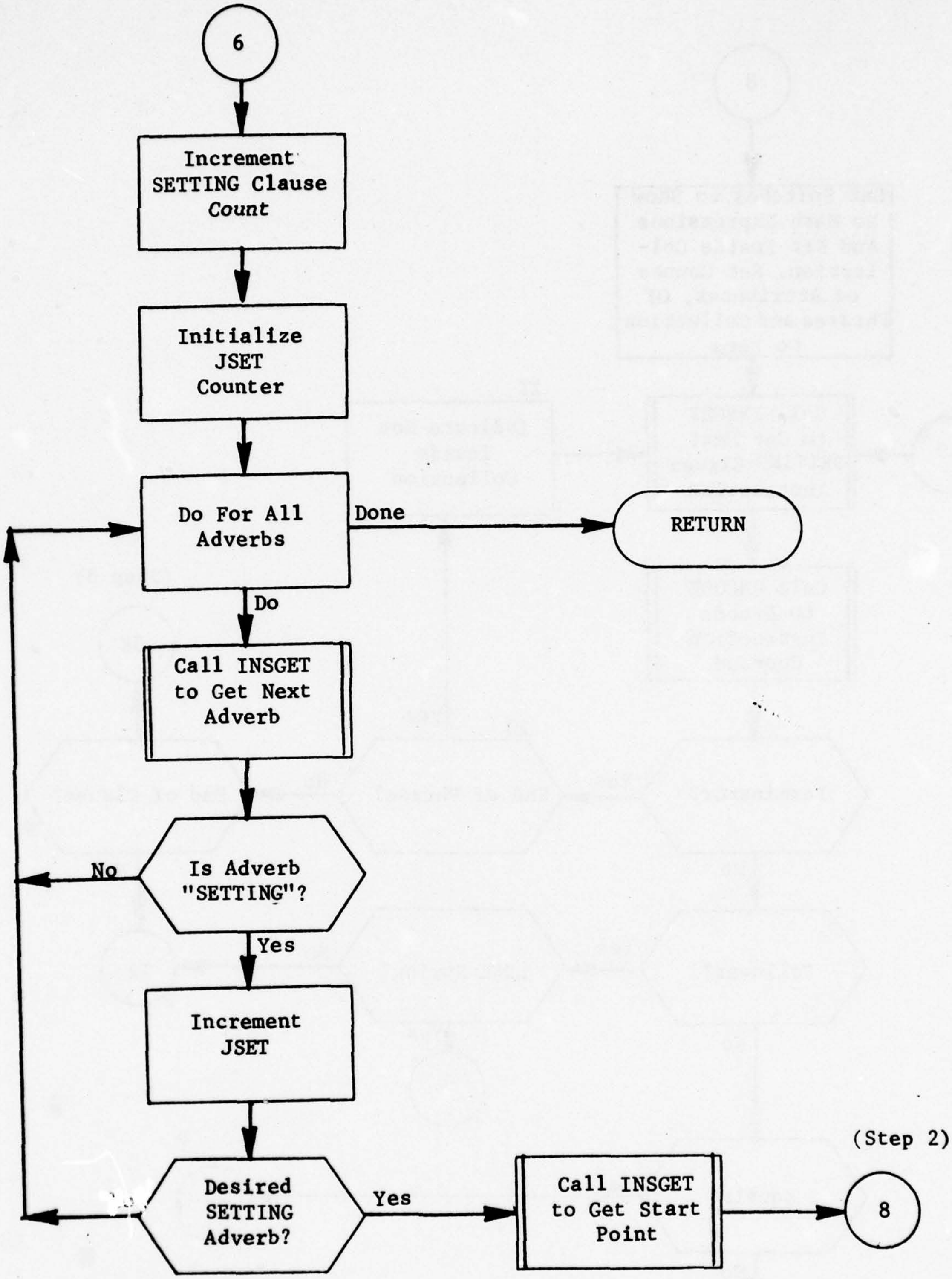



Figure 53. (Part 3 of 3)

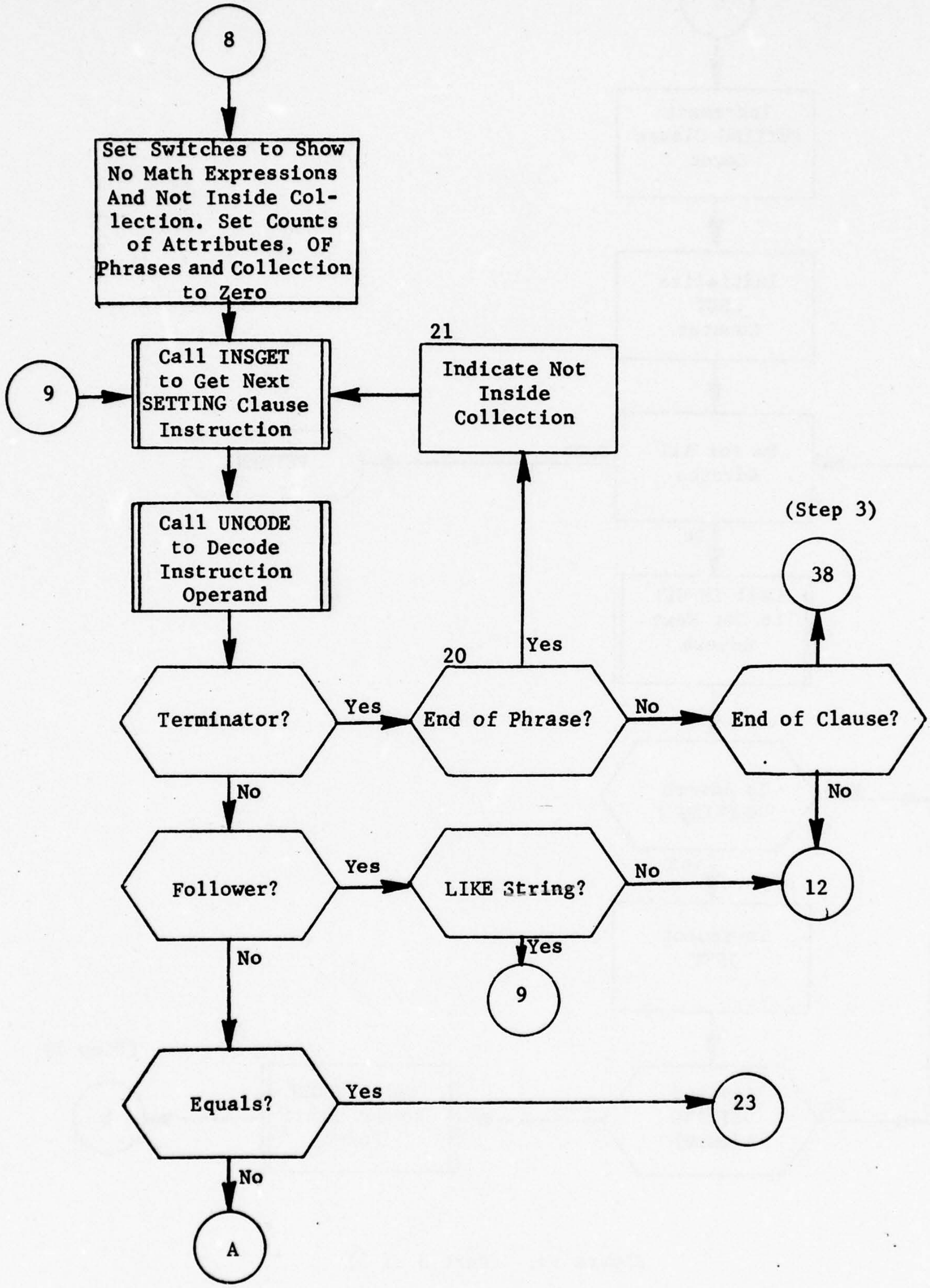



Figure 54. Subroutine CREAAT: Step 2 (Part 1 of 6)

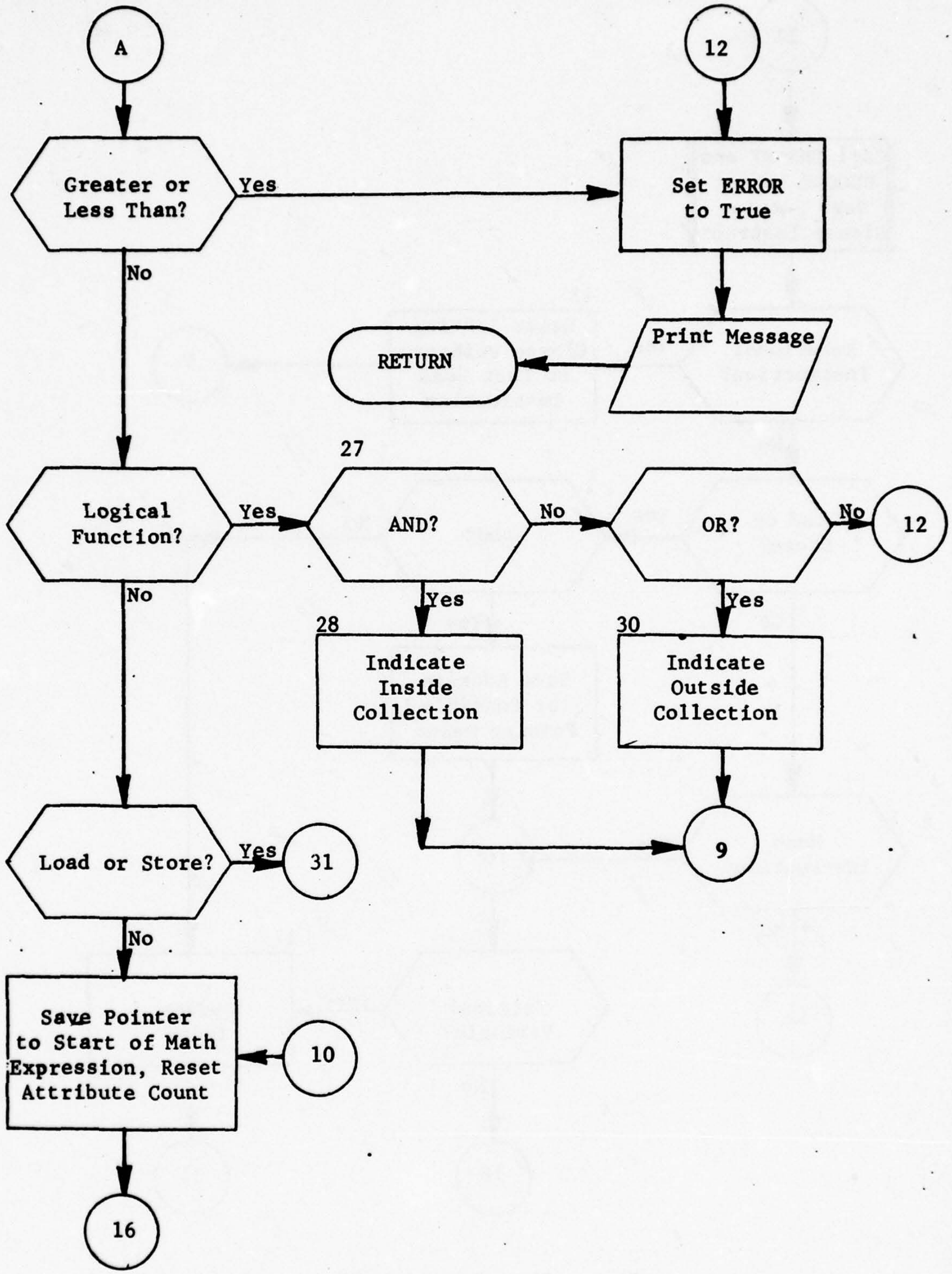



Figure 54. (Part 2 of 6)

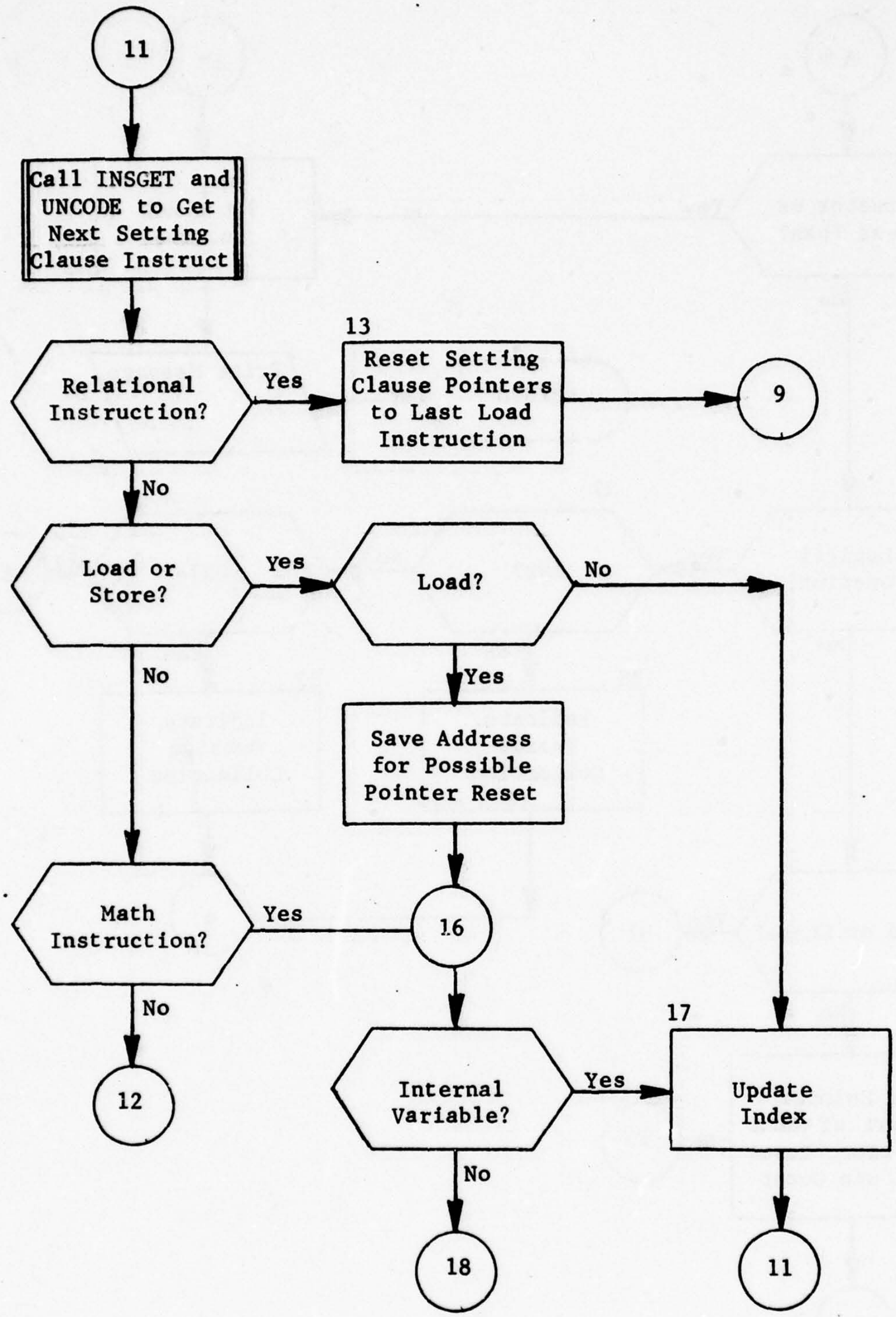



Figure 54. (Part 3 of 6)

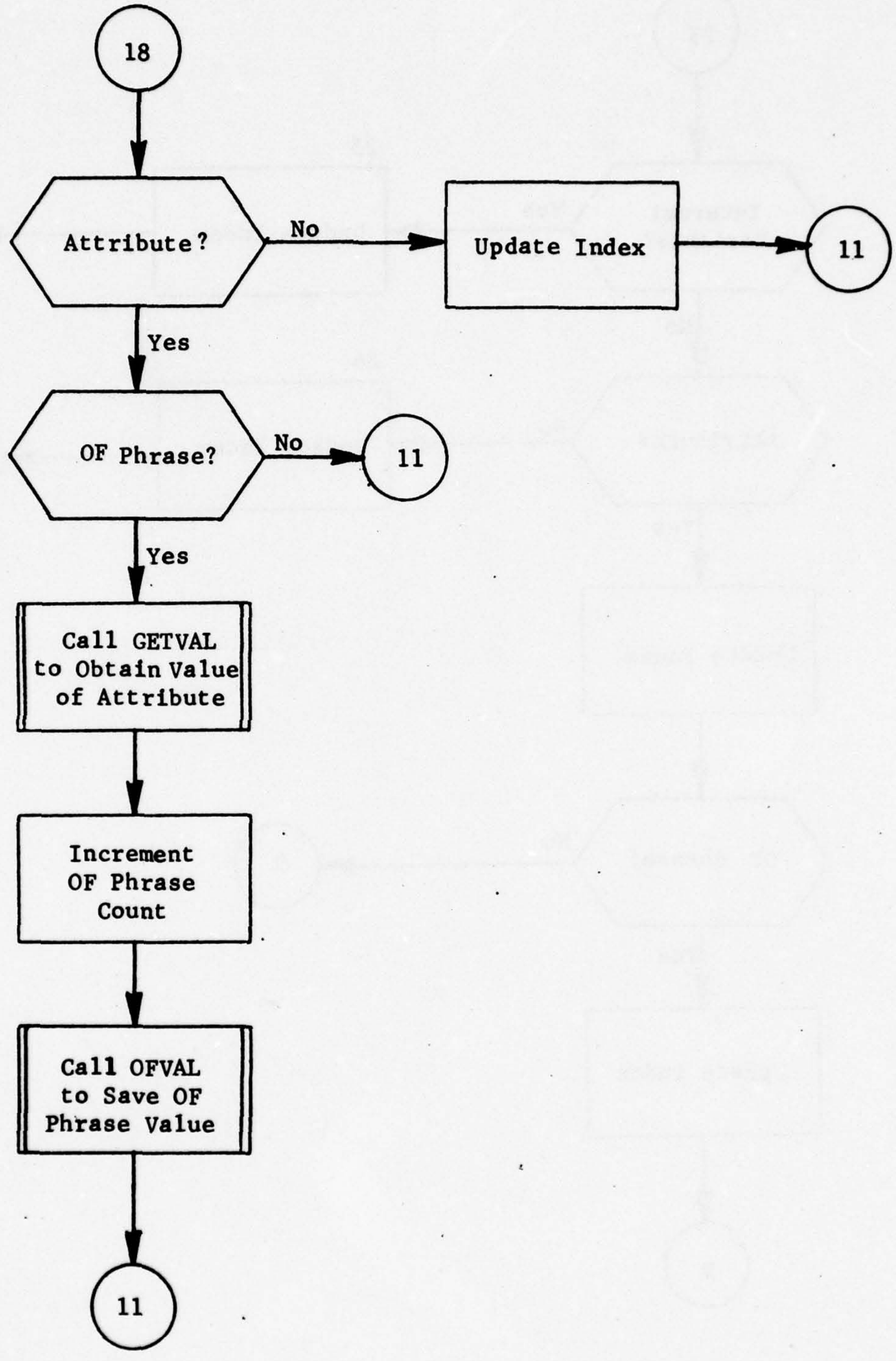



Figure 54. (Part 4 of 6)

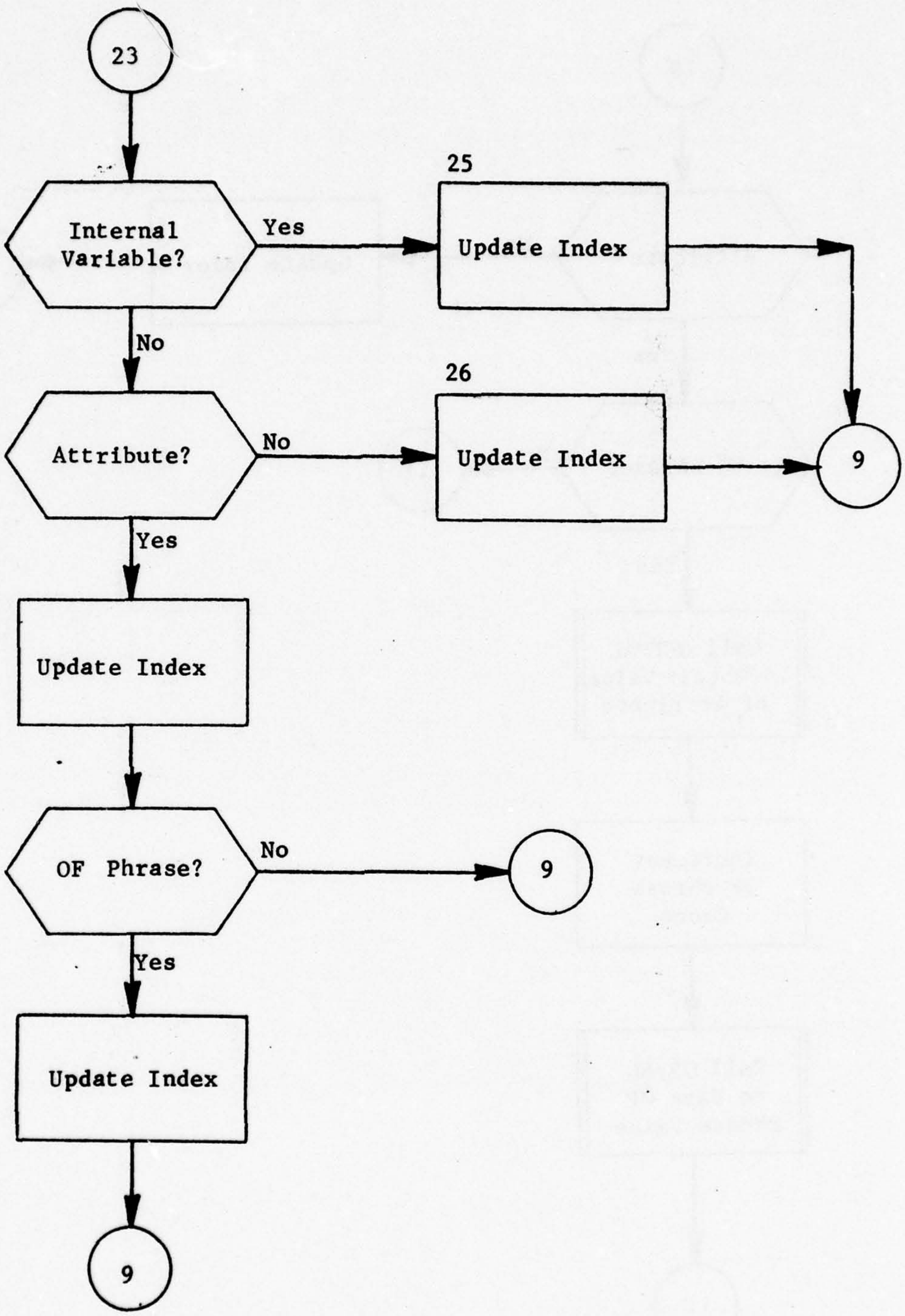



Figure 54. (Part 5 of 6)

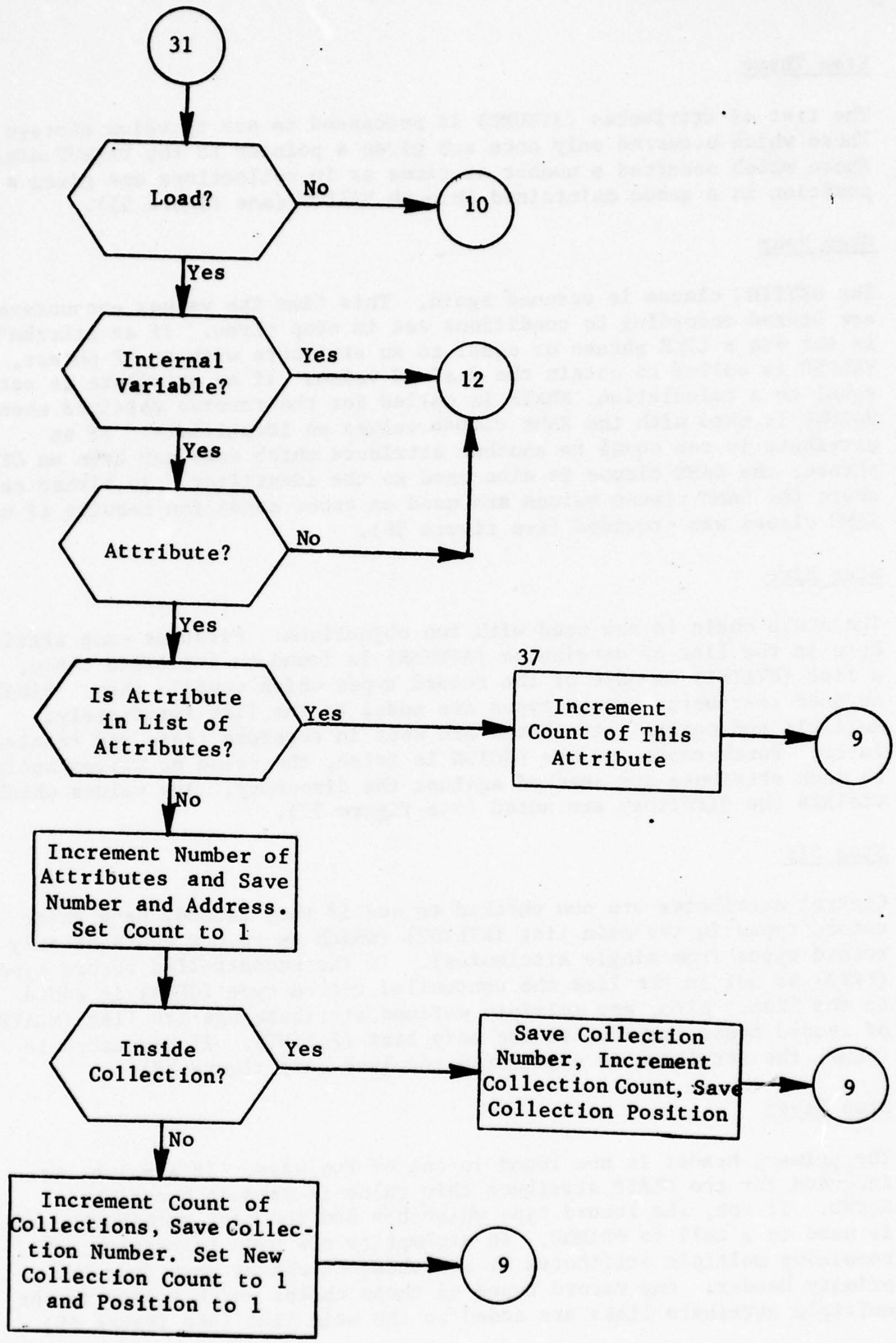



Figure 54. (Part 6 of 6)

Step Three

The list of attributes (ATNUMB) is processed to set up value storage. Those which occurred only once are given a pointer to the VALBUF array. Those which occurred a number of times or in collections are given a position in a queue maintained through VALPUT (see figure 55).

Step Four

The SETTING clause is scanned again. This time the values encountered are stored according to conditions set in step three. If an attribute is set via a LIKE phrase or equal to an attribute with an OF phrase, VALFND is called to obtain the desired value. If an attribute is set equal to a calculation, XMATH is called for the records obtained when VALFND is used with the SAME clause values as identifiers. If an attribute is set equal to another attribute which does not have an OF phrase, the SAME clause is also used as the identifier. In either case where the SAME clause values are used an error condition results if no SAME clause was provided (see figure 56).

Step Five

The ATRIB chain is now used with two objectives. First as each attribute in the list of attributes (ATNUMB) is found on the ATRIB chain, a list (RTLIST) is made of the record types which contain them. Single defined attributes record types are added to the list immediately. Multiple and control attributes are kept in separate lists and resolved later. Furthermore, unless DRCTSW is false, the value or values assigned to each attribute are checked against the directory. Any values which violate the directory are noted (see figure 57).

Step Six

Control attributes are now checked to see if they already have their record types in the main list (RTLIST) (which up to now contains only record types from single attributes). If the uncontrolled record type (CNTB) is not in the list the controlled record type (CNTA) is added to the list. Also, any multiply defined attribute has its list (MLATPT) of record types compared to the main list (RTLIST). If any match is found, the attribute is considered resolved (see figure 58).

Step Seven

The primary header is now found in one of two ways. If a value was included for the CLASS attribute this value is used in a call to HDFND. If not, the record type which has had the most attributes set is used in a call to PRIMHD. An attempt is now made to resolve any remaining multiple attributes by searching on chains down from the primary header. Any record types on these chains which appear in the multiple attribute lists are added to the main list (see figure 59).

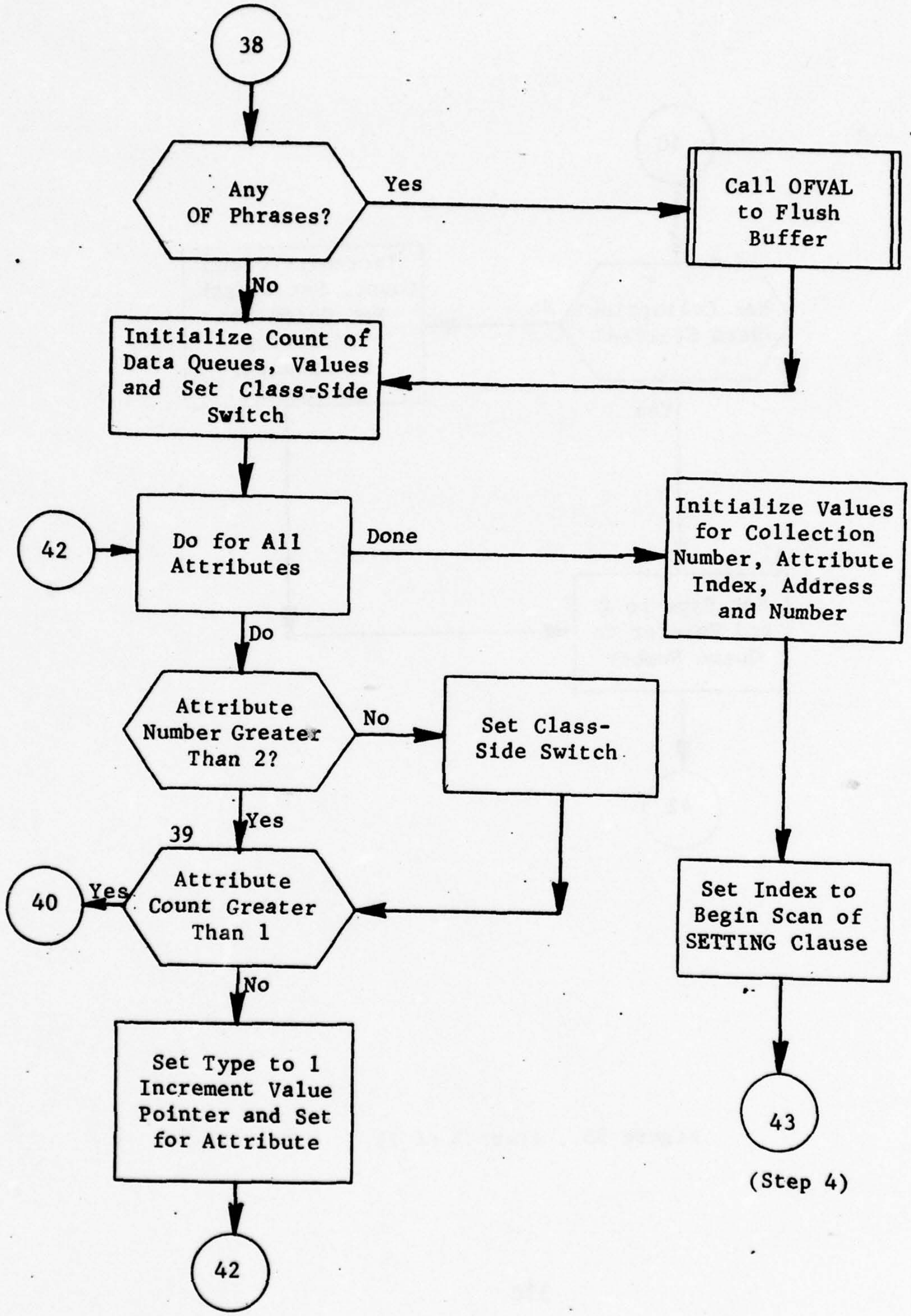



Figure 55. Subroutine CREAAT: Step 3 (Part 1 of 2)

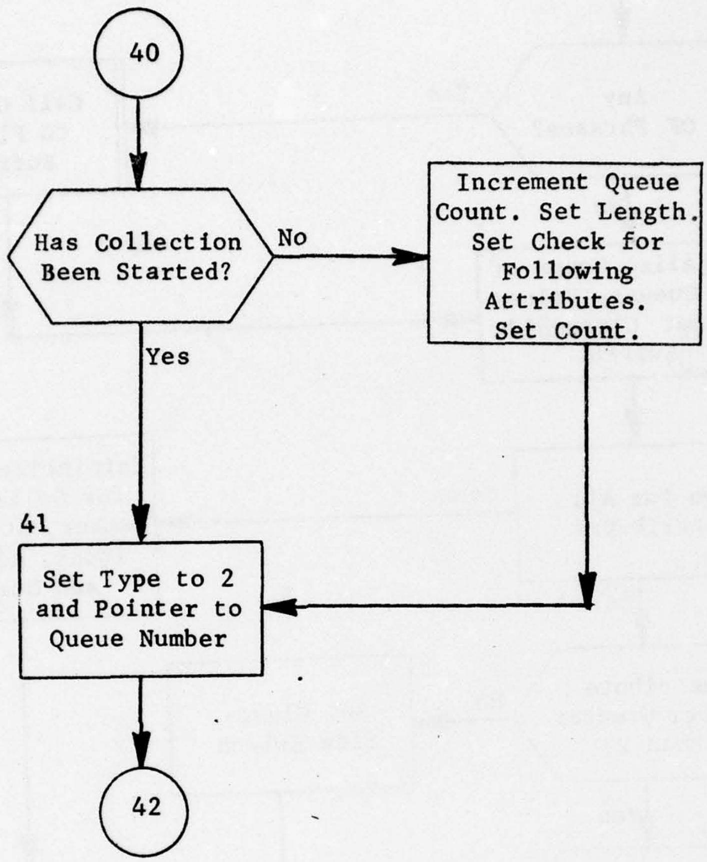



Figure 55. (Part 2 of 2)

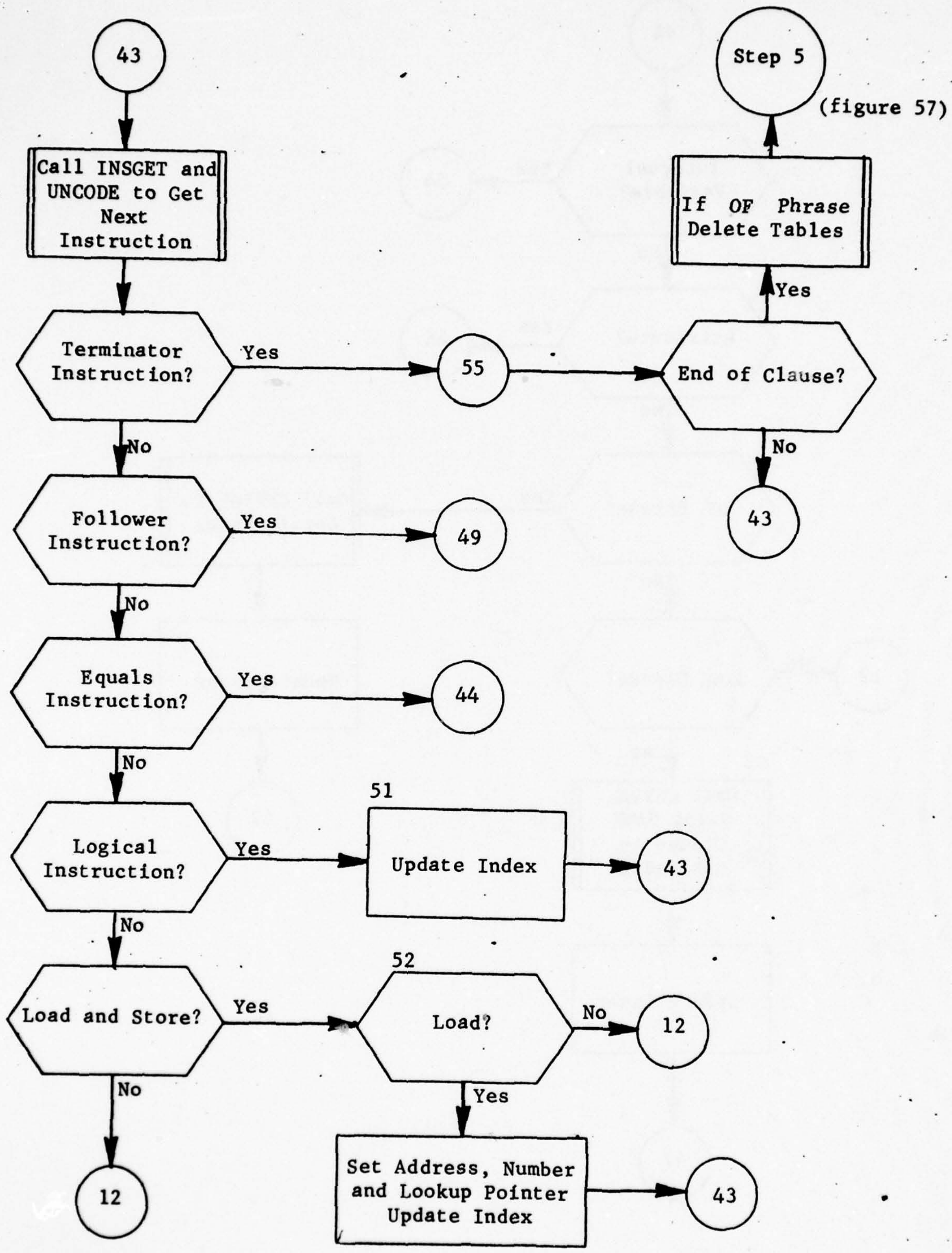



Figure 56. Subroutine CREAAT: Step 4 (Part 1 of 5)

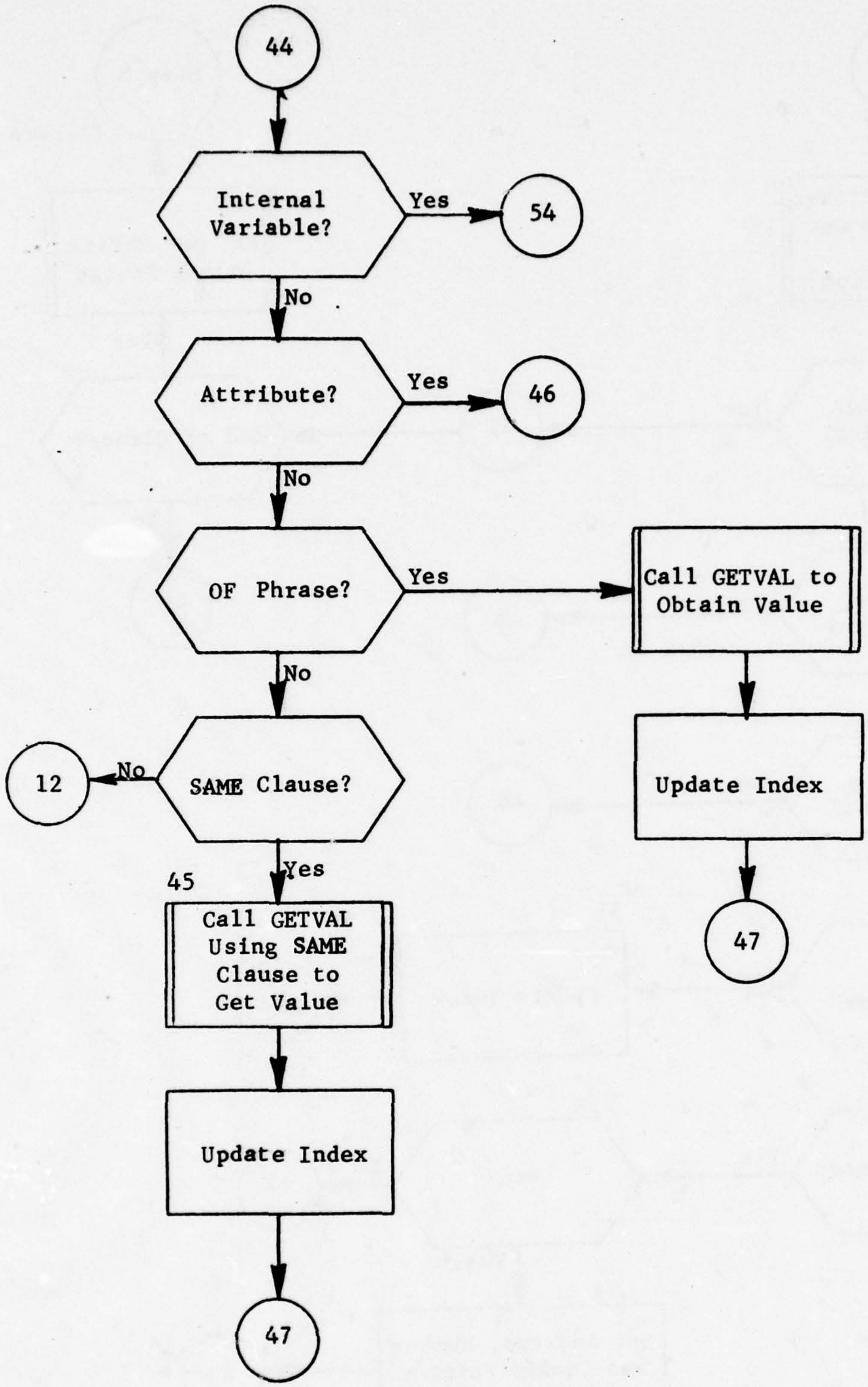



Figure 56. (Part 2 of 5)

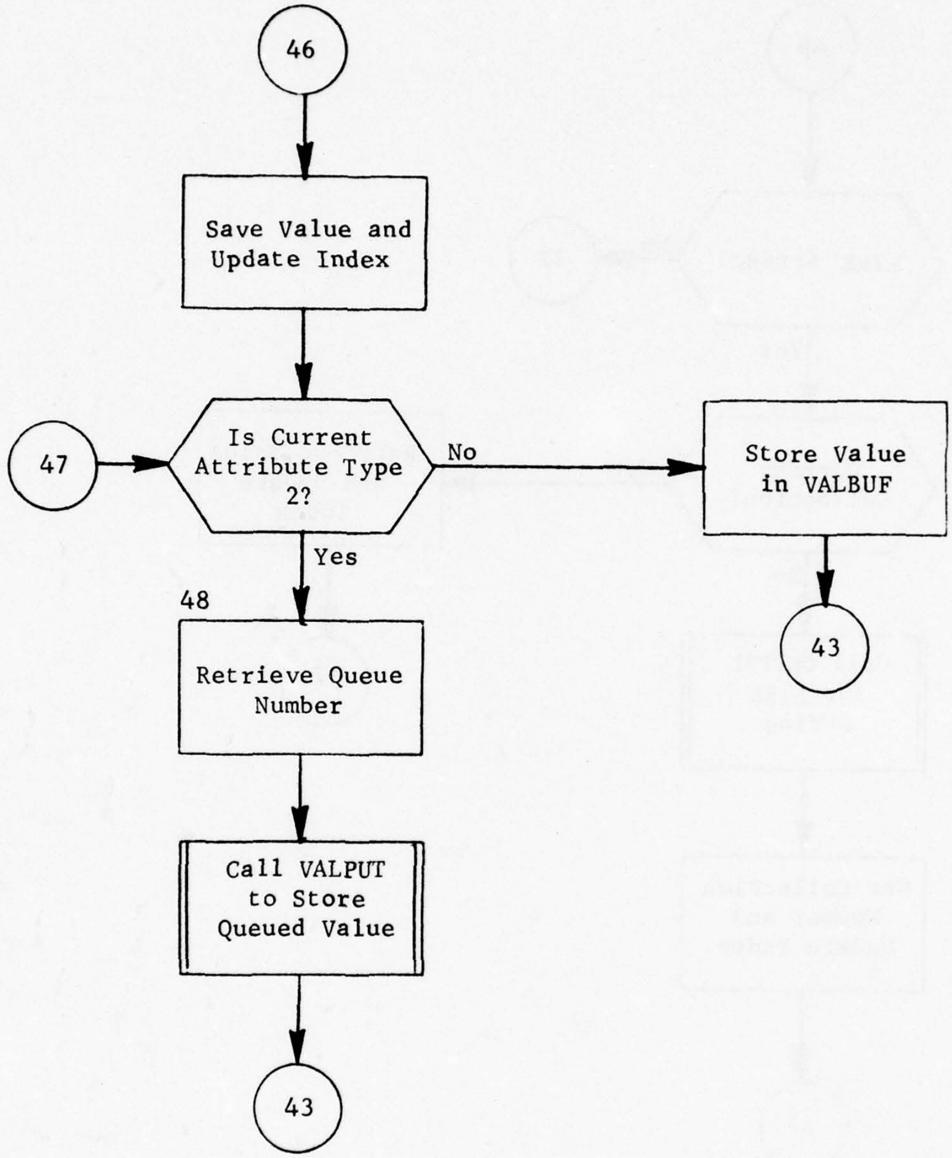



Figure 56. (Part 3 of 5)

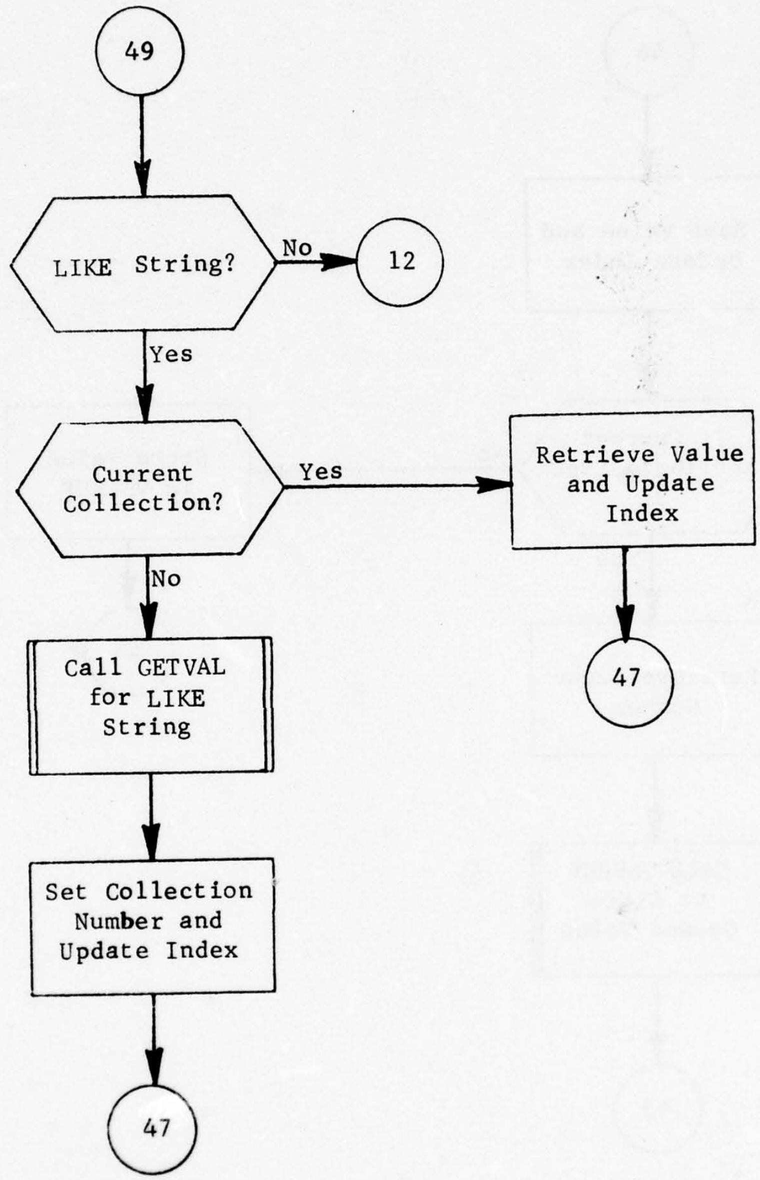



Figure 56. (Part 4 of 5)

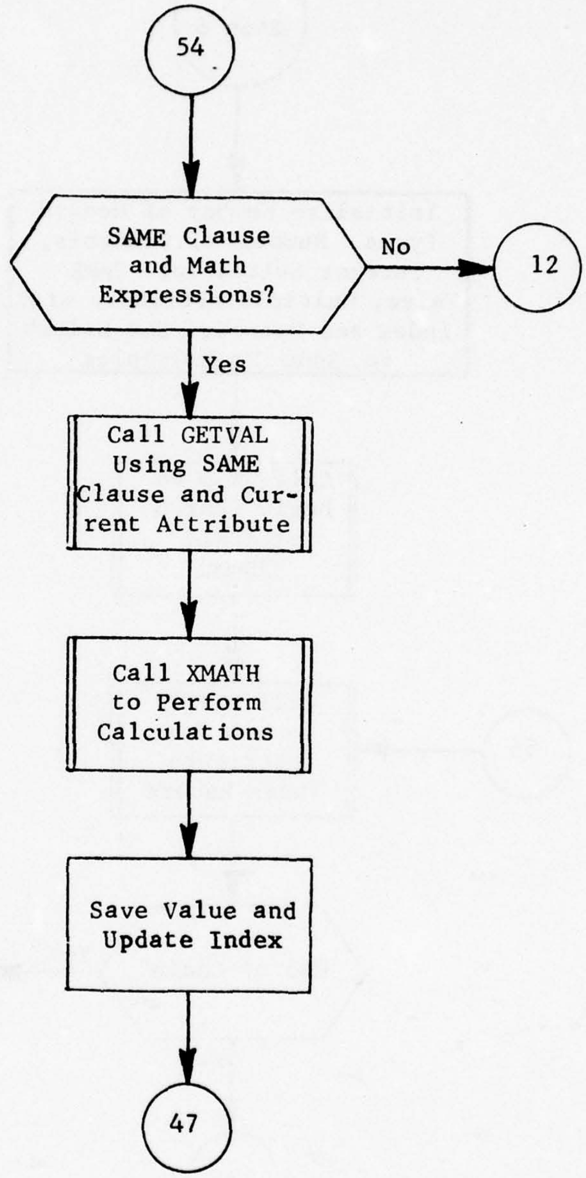



Figure 56. (Part 5 of 5)

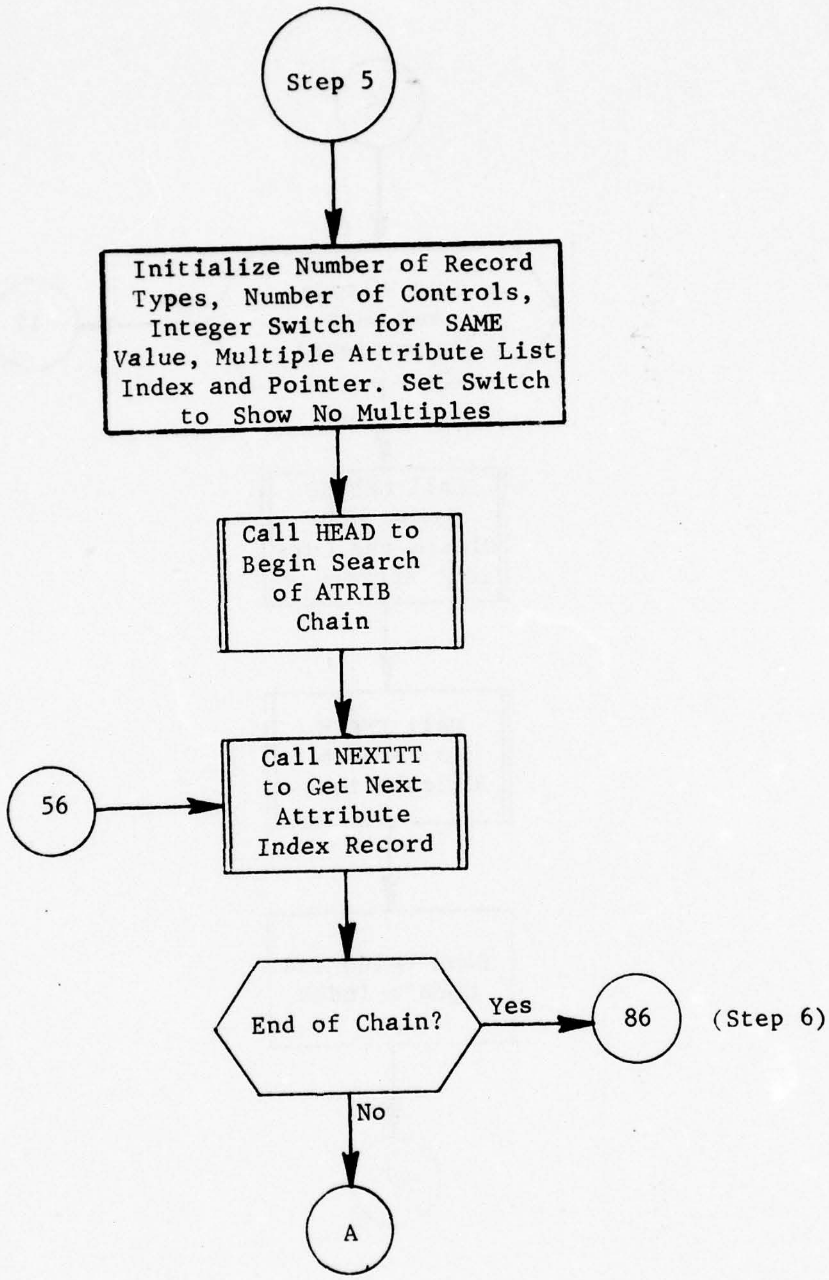



Figure 57. Subroutine CREAAT: Step 5 (Part 1 of 7)

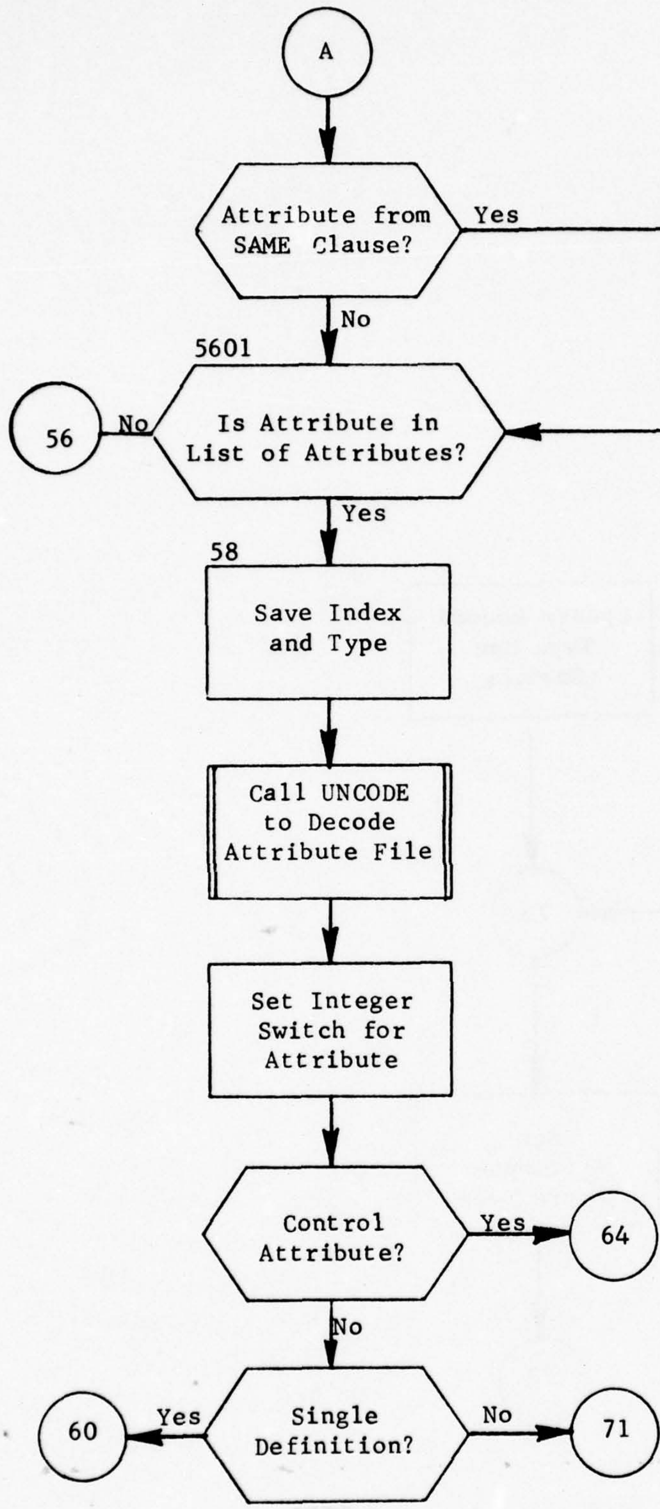



Figure 57. (Part 2 of 7)

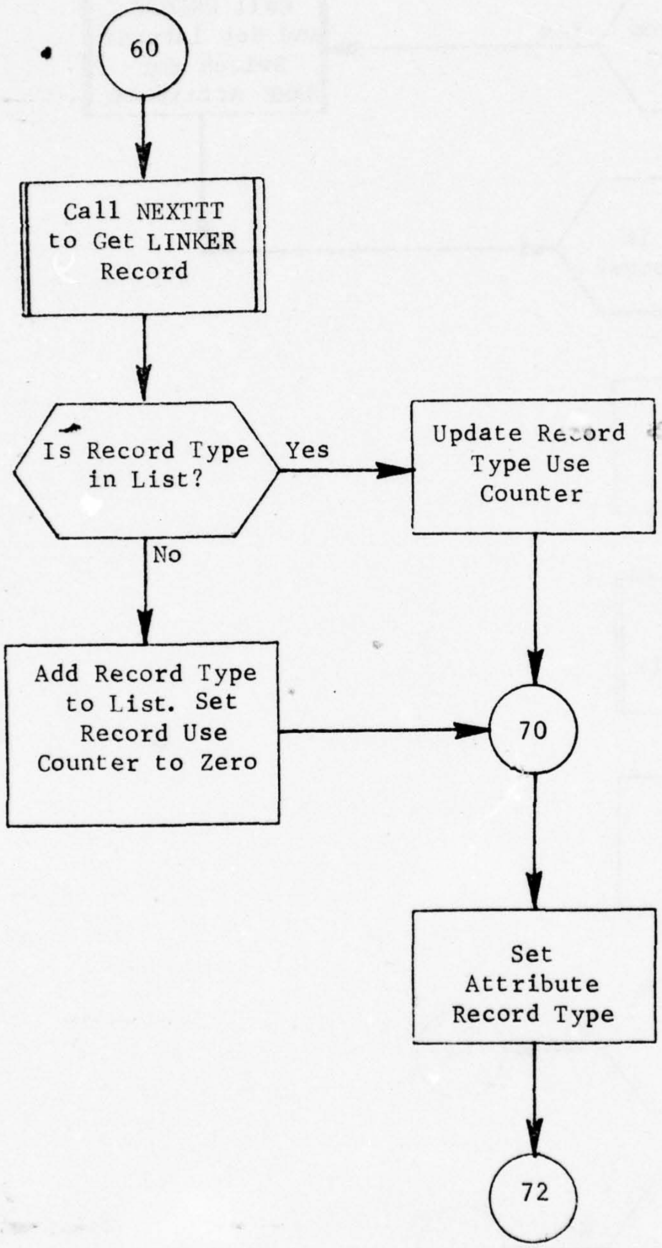



Figure 57. (Part 3 of 7)

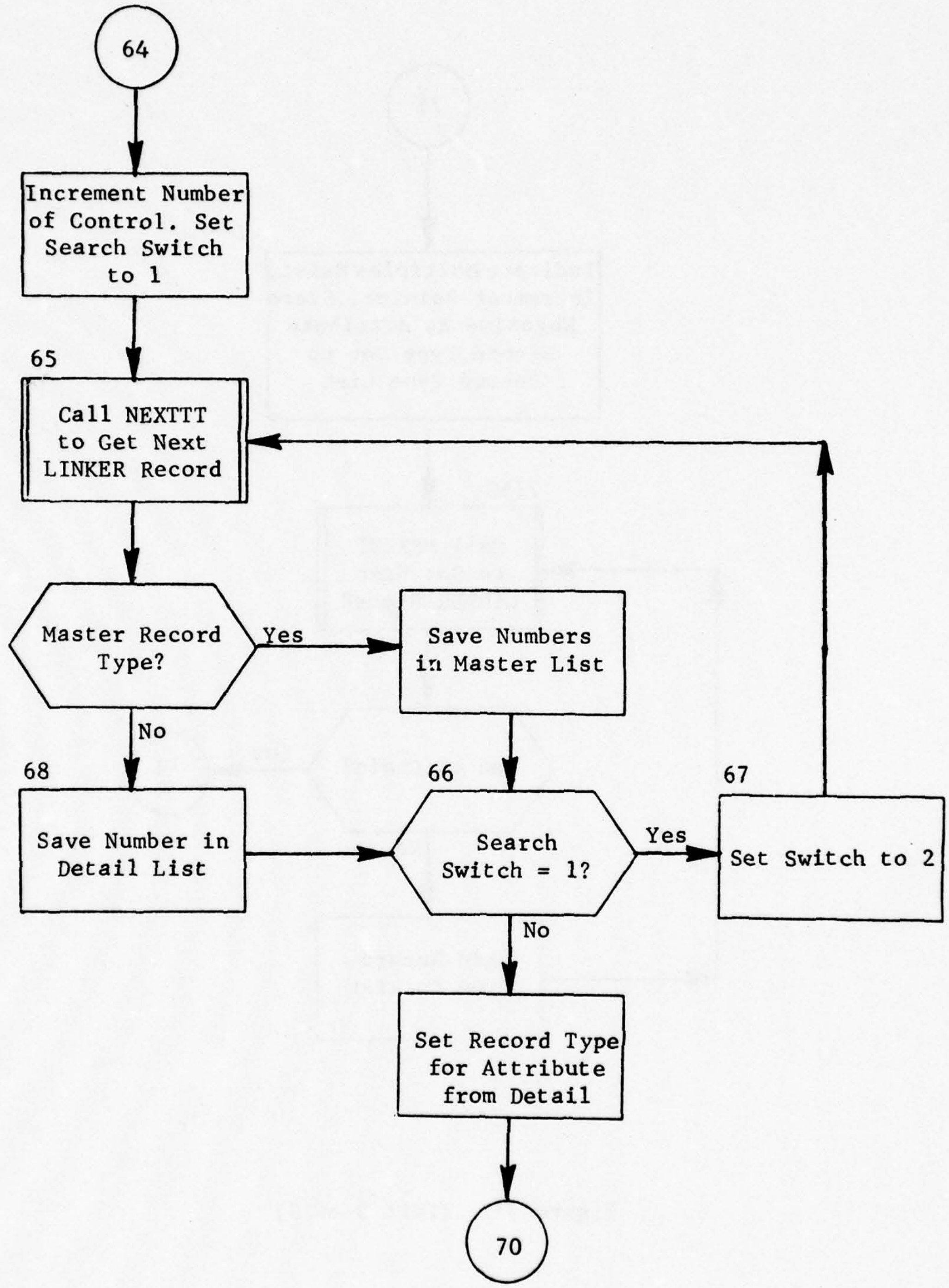



Figure 57. (Part 4 of 7)

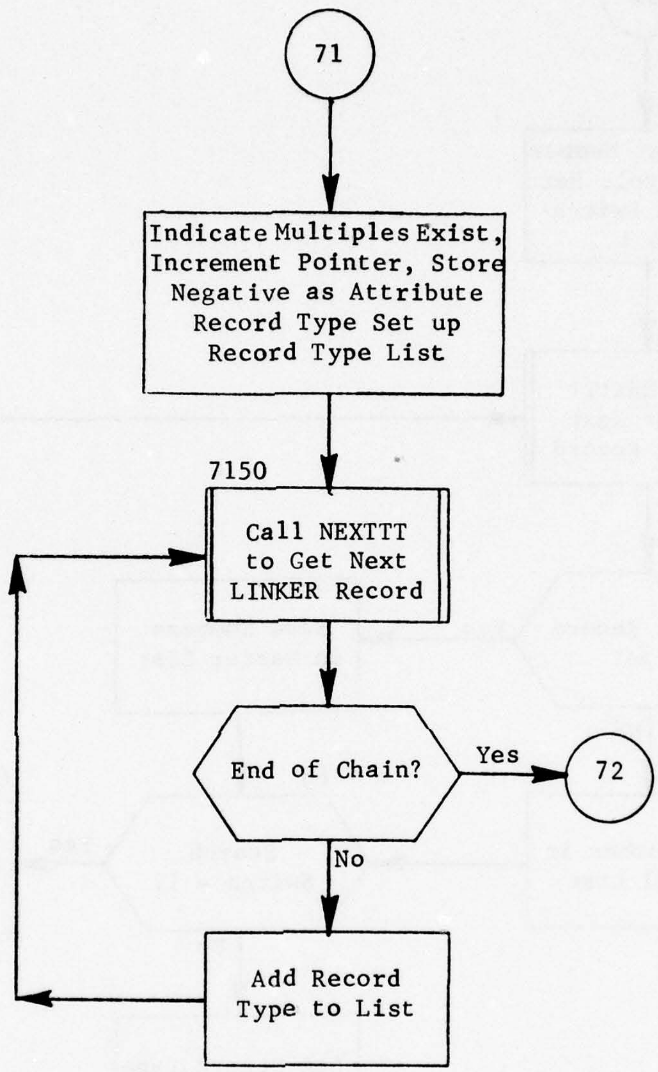



Figure 57. (Part 5 of 7)

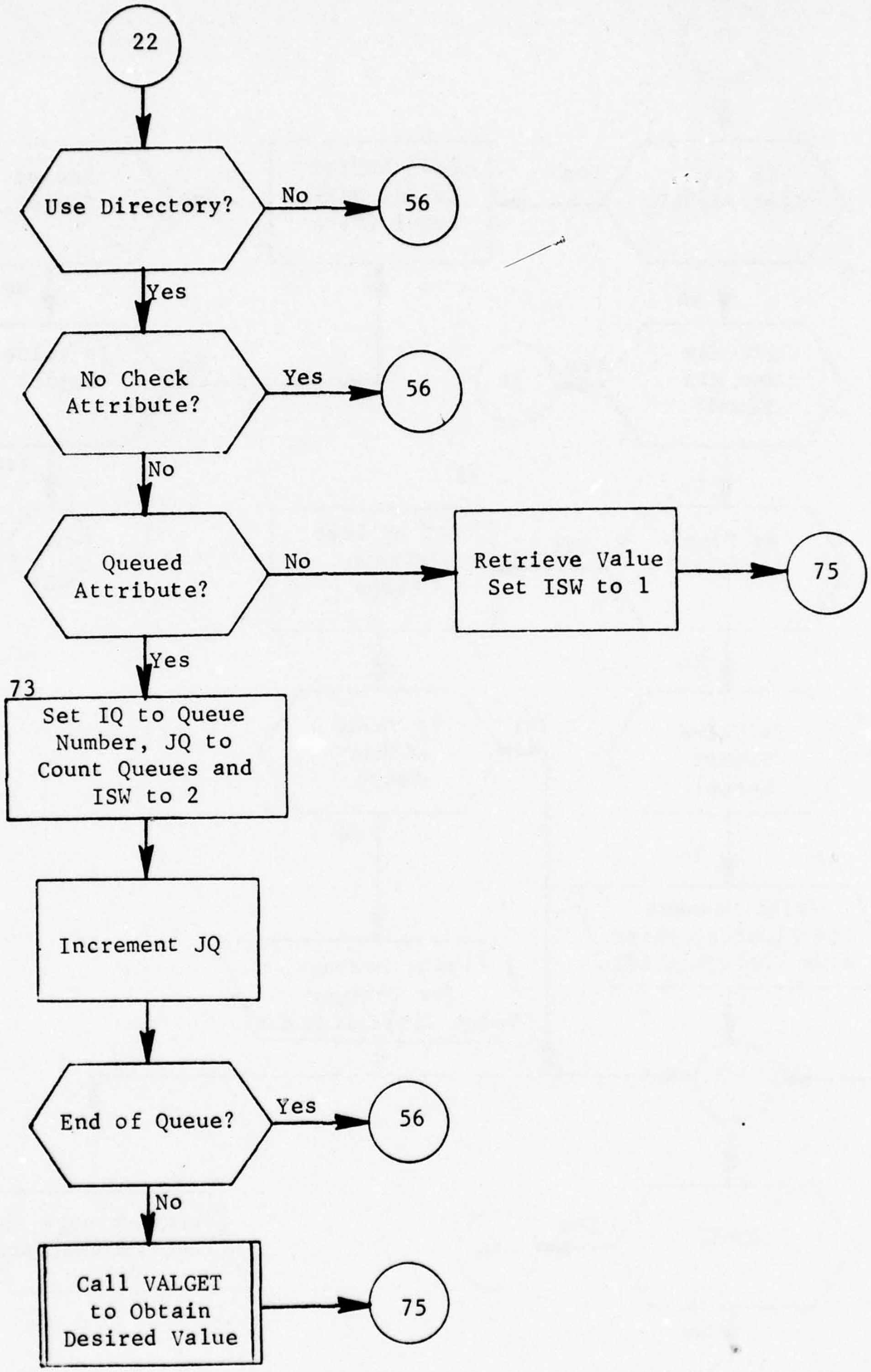



Figure 57. (Part 6 of 7)

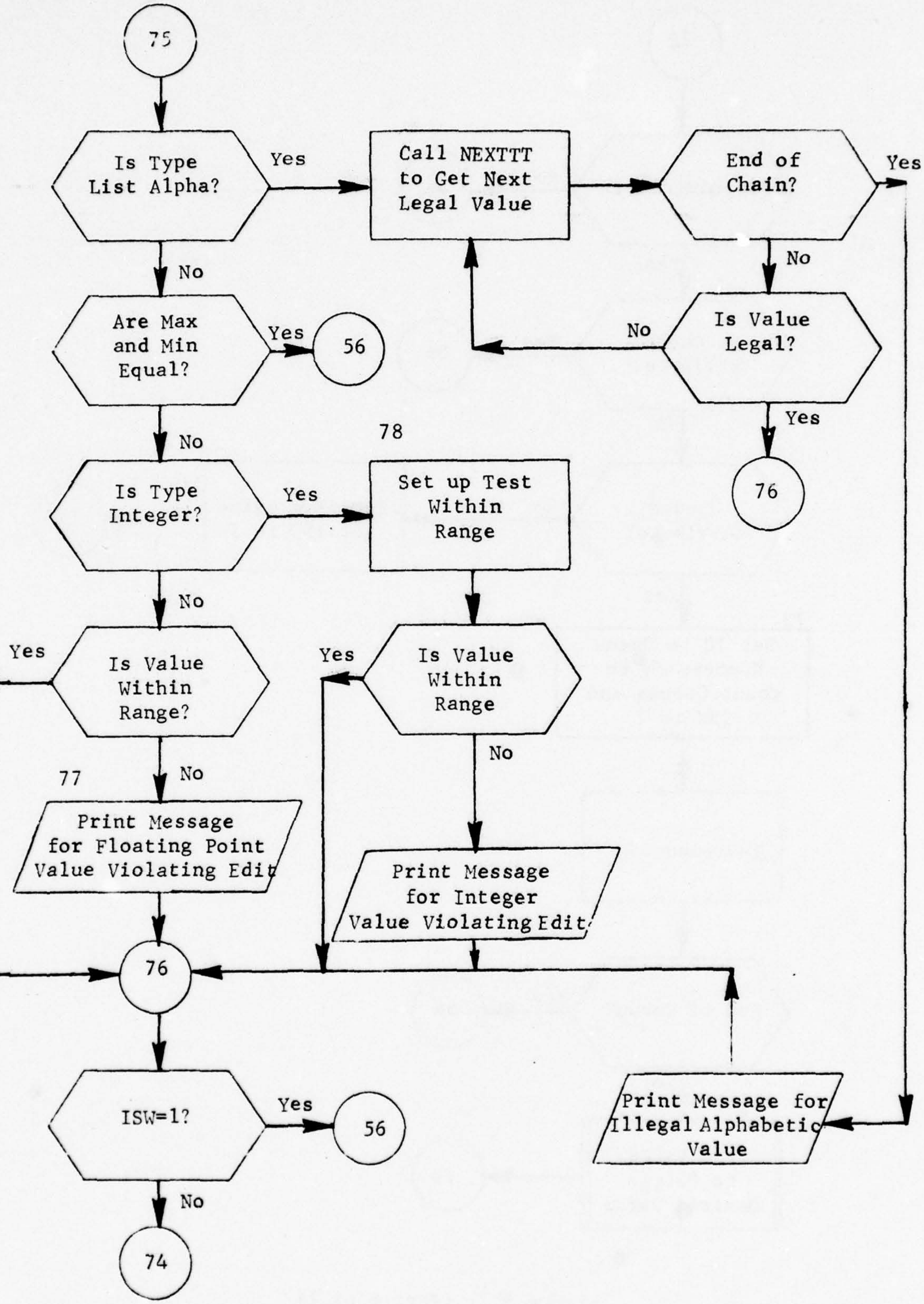



Figure 57. (Part 7 of 7)

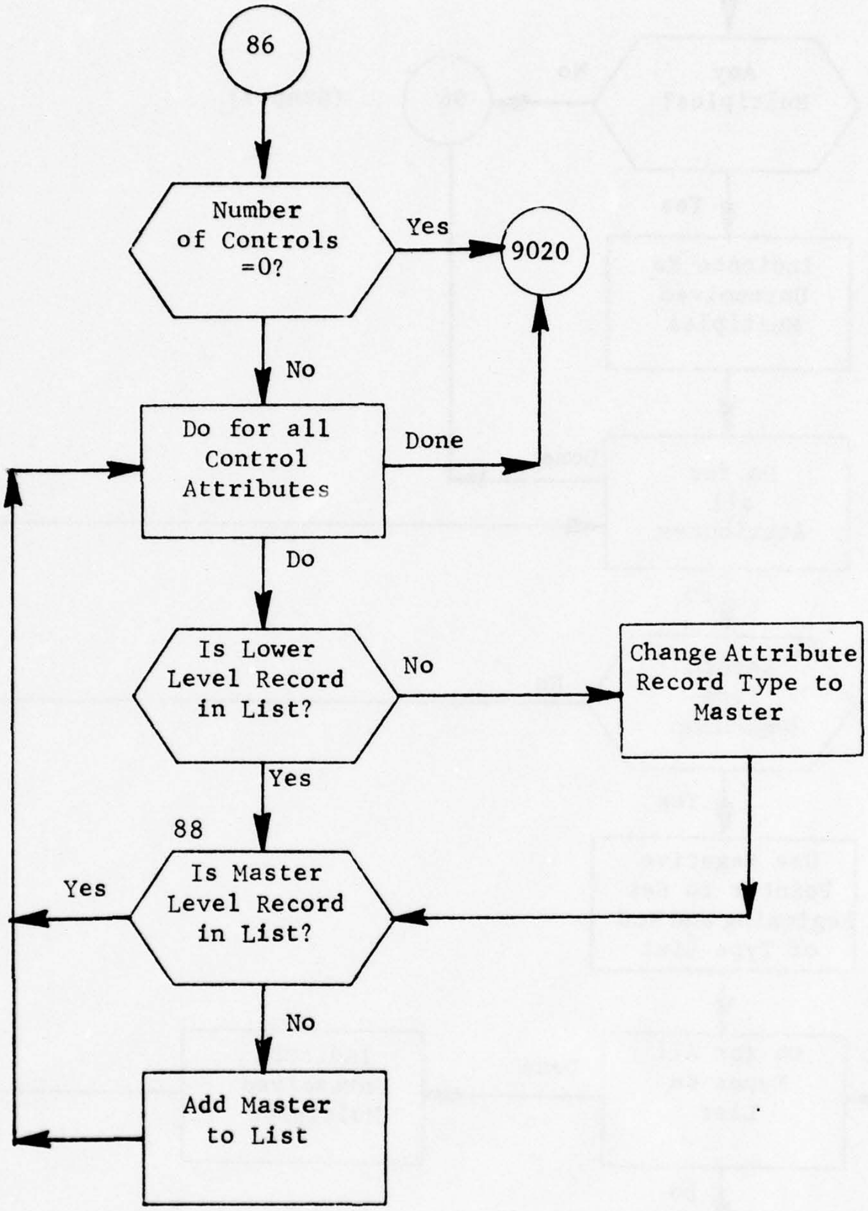



Figure 58. Subroutine CREAAT: Step 6 (Part 1 of 2)

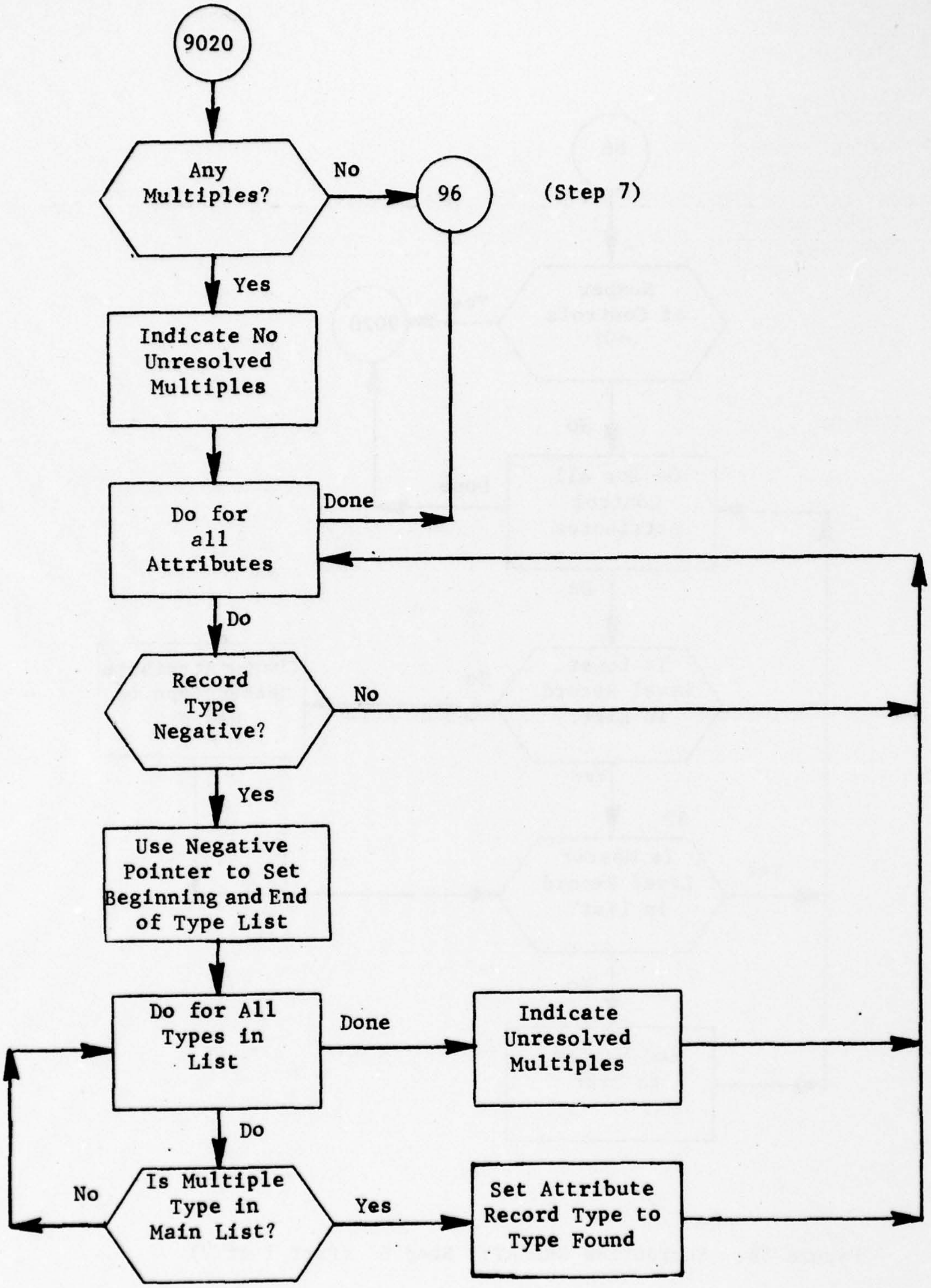



Figure 58. (Part 2 of 2)

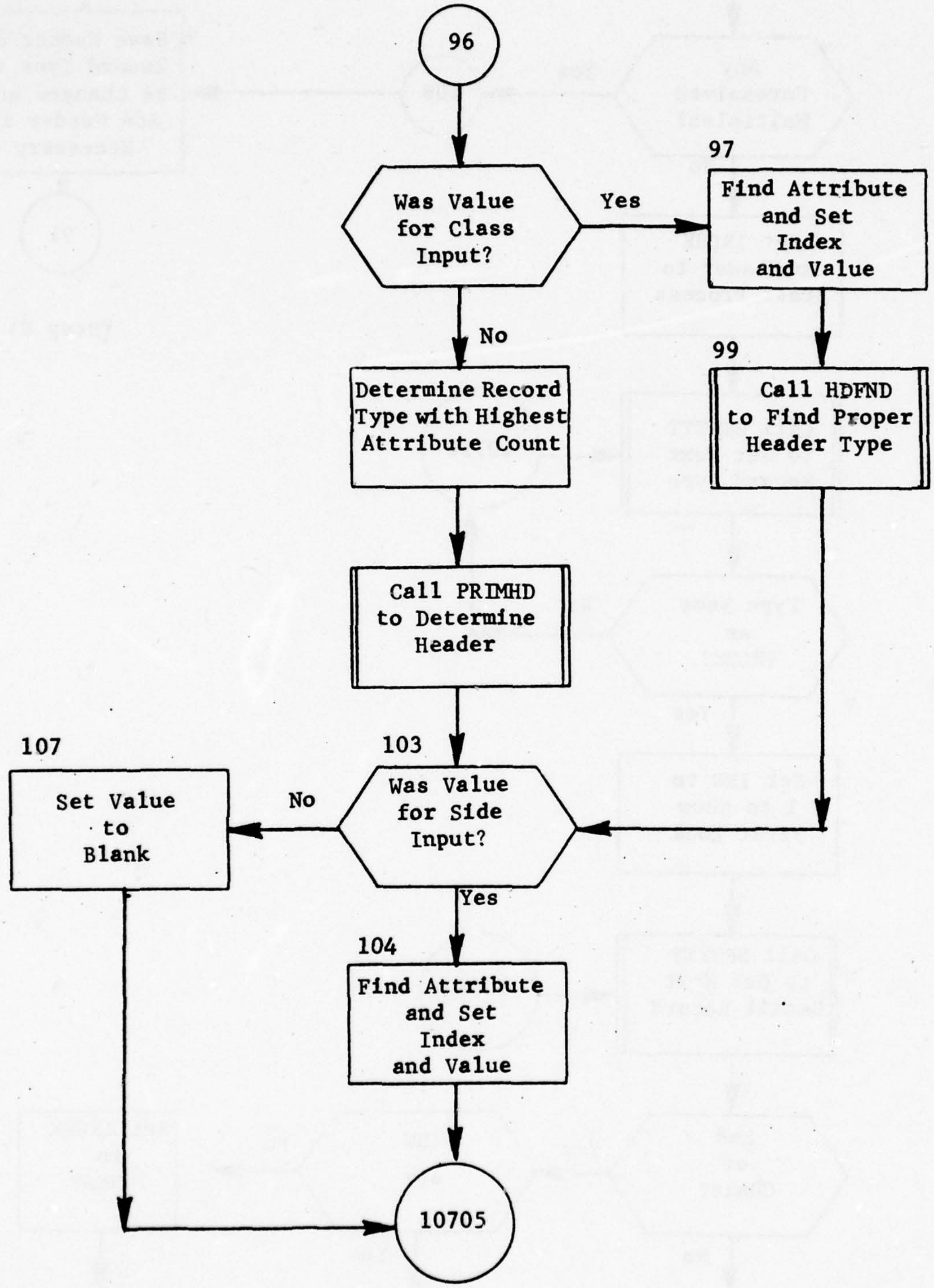



Figure 59. Subroutine CREAAT: Step 7 (Part 1 of 3)

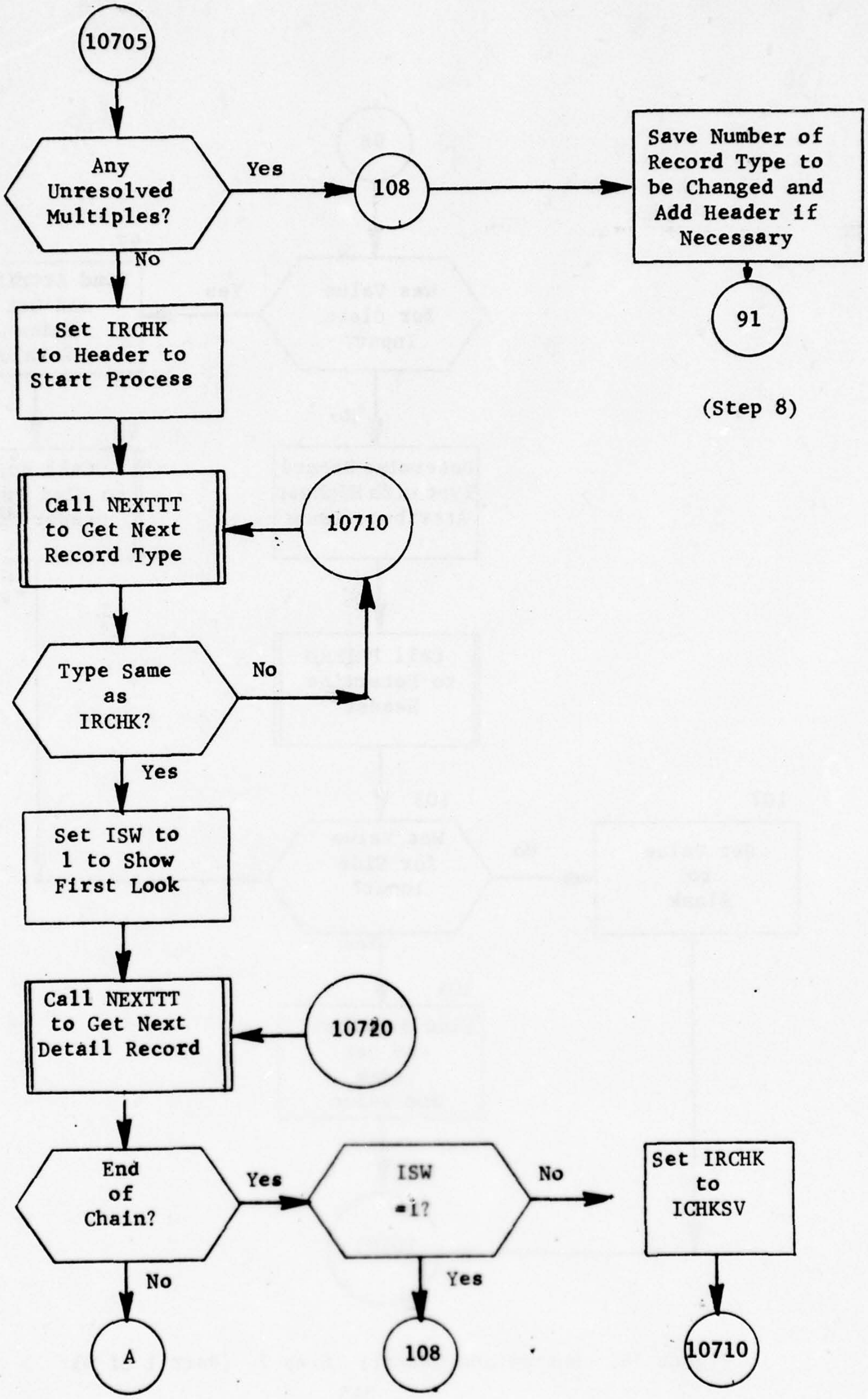



Figure 59. (Part 2 of 3)

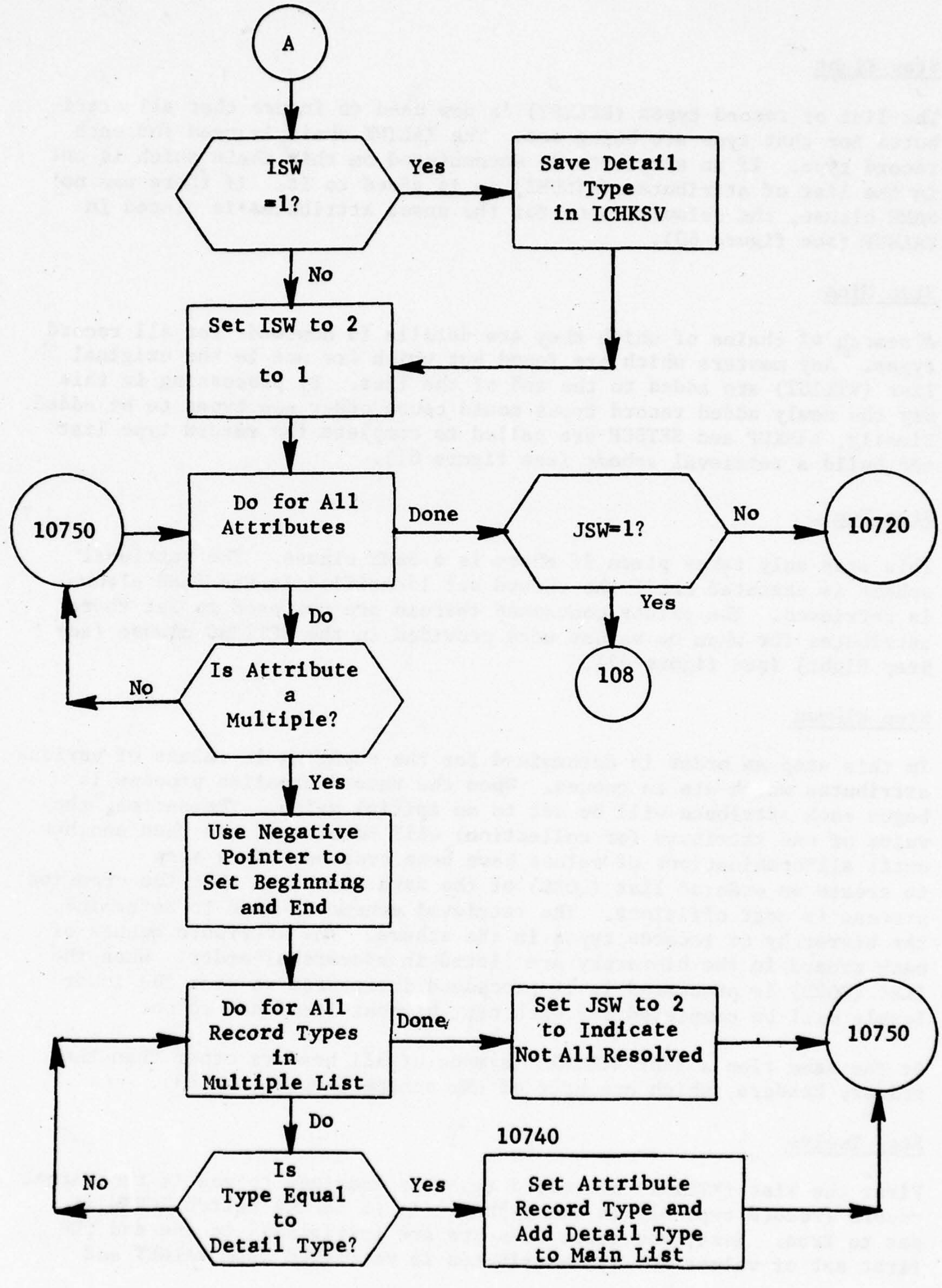



Figure 59. (Part 3 of 3)

Step Eight

The list of record types (RTLIST) is now used to insure that all attributes for that type are being set. The IALINK chain is used for each record type. If an attribute is encountered on this chain which is not in the list of attributes (ATNUMB) is is added to it. If there was no SAME clause, the default value for the unset attributes is placed in VALBUF (see figure 60).

Step Nine

A search of chains of which they are details is now made for all record types. Any masters which are found but which are not in the original list (RTLIST) are added to the end of the list. By processing in this way the newly added record types could cause other new types to be added. Finally, LINKUP and SETSCH are called to complete the record type list and build a retrieval scheme (see figure 61).

Step Ten

This step only takes place if there is a SAME clause. The retrieval scheme is executed until the record set identified in the SAME clause is retrieved. The values contained therein are now used to set those attributes for whom no values were provided in the SETTING clause (see Step Eight) (see figure 62).

Step Eleven

In this step an order is determined for the changing in values of various attributes which are in queues. When the record creation process is begun each attribute will be set to an initial value. Thereafter, the value of one attribute (or collection) will be altered and then another until all combinations of values have been created. This step to create an ordered list (QORD) of the data queues so that the creation process is most efficient. The retrieval scheme is used to determine the hierarchy of records types in the scheme. The attribute queues of each record in the hierarchy are listed in hierarchal order. When the list (QORD) is processed it is processed in reverse so that the lower levels will be completed for each next highest level and so on.

At the same time a list (OUTHD) is made of all headers other than the primary headers, which are part of the scheme (see figure 63).

Step Twelve

First the list (RTLIST) of record types is examined to see if the target record (record type 61) is among them. If it is, the switch TGTSW is set to true. Next, the queue counters are initialized to one and the first set of values for all attributes is retrieved from VALGET and

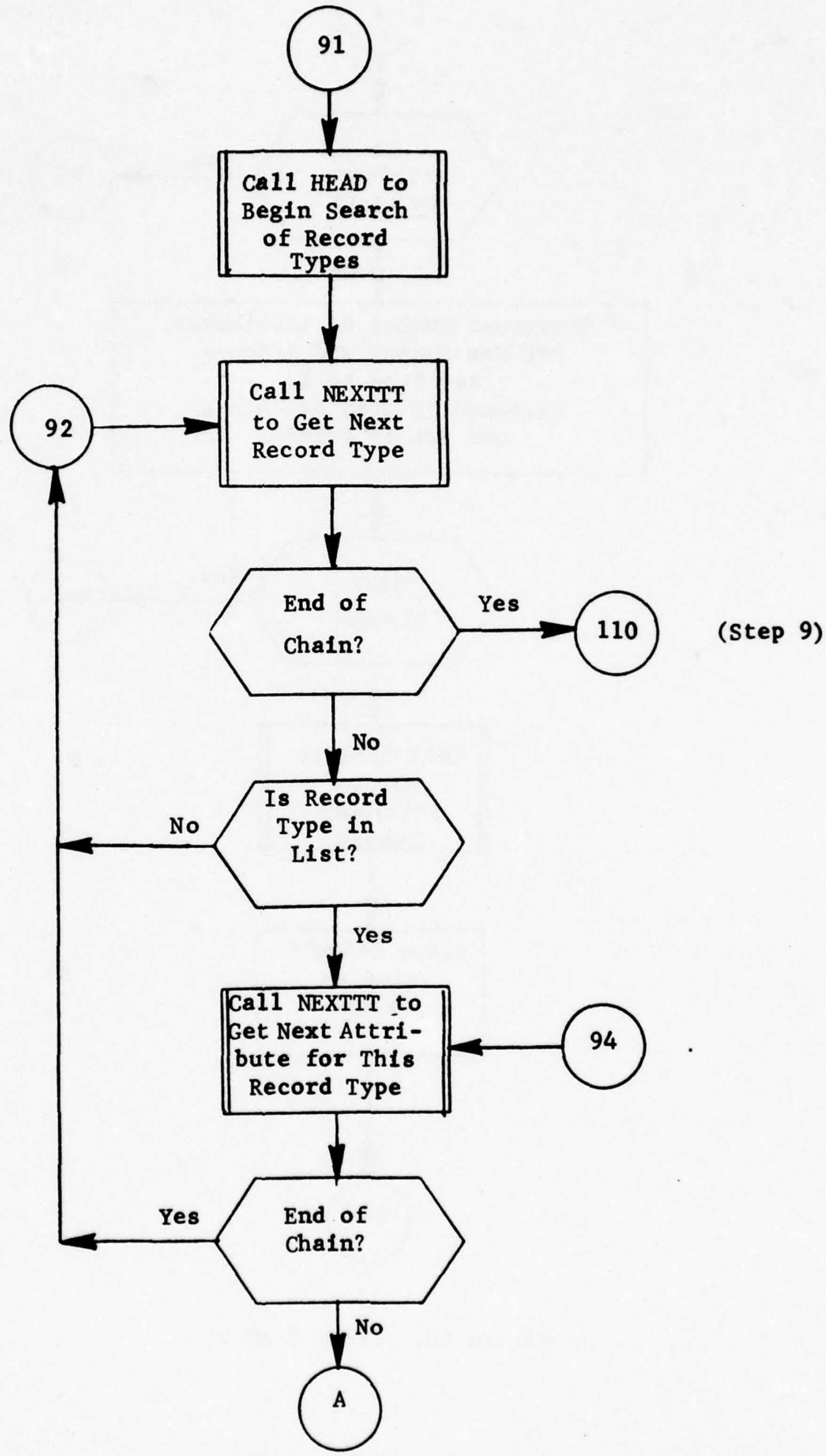



Figure 60. Subroutine CREAAT: Step 8 (Part 1 of 2)

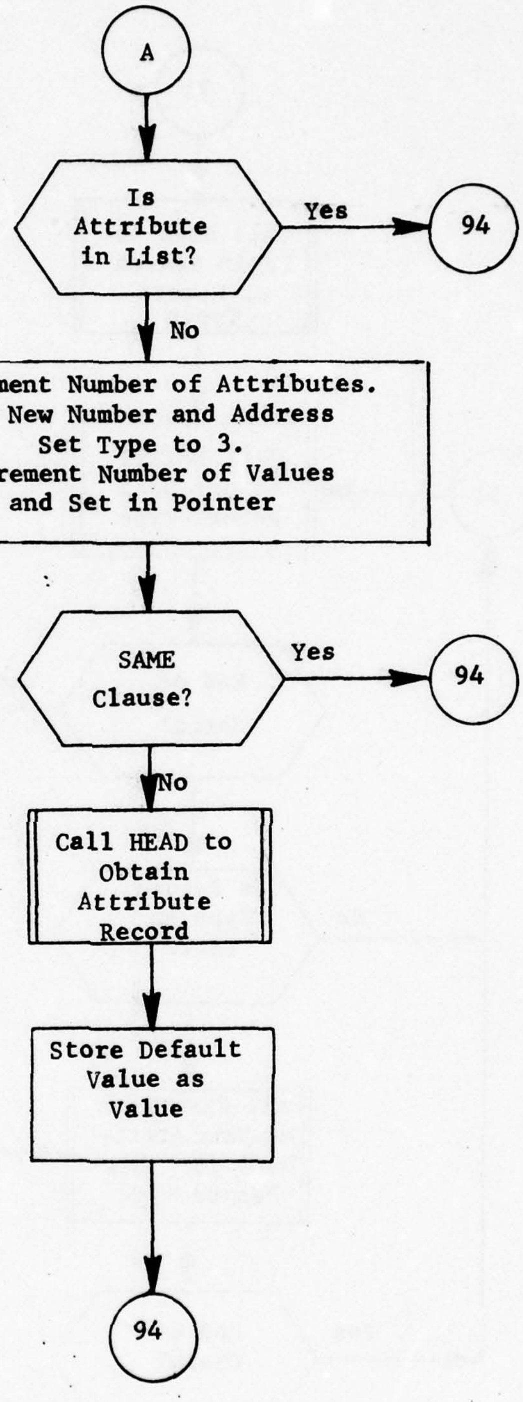



Figure 60. (Part 2 of 2)

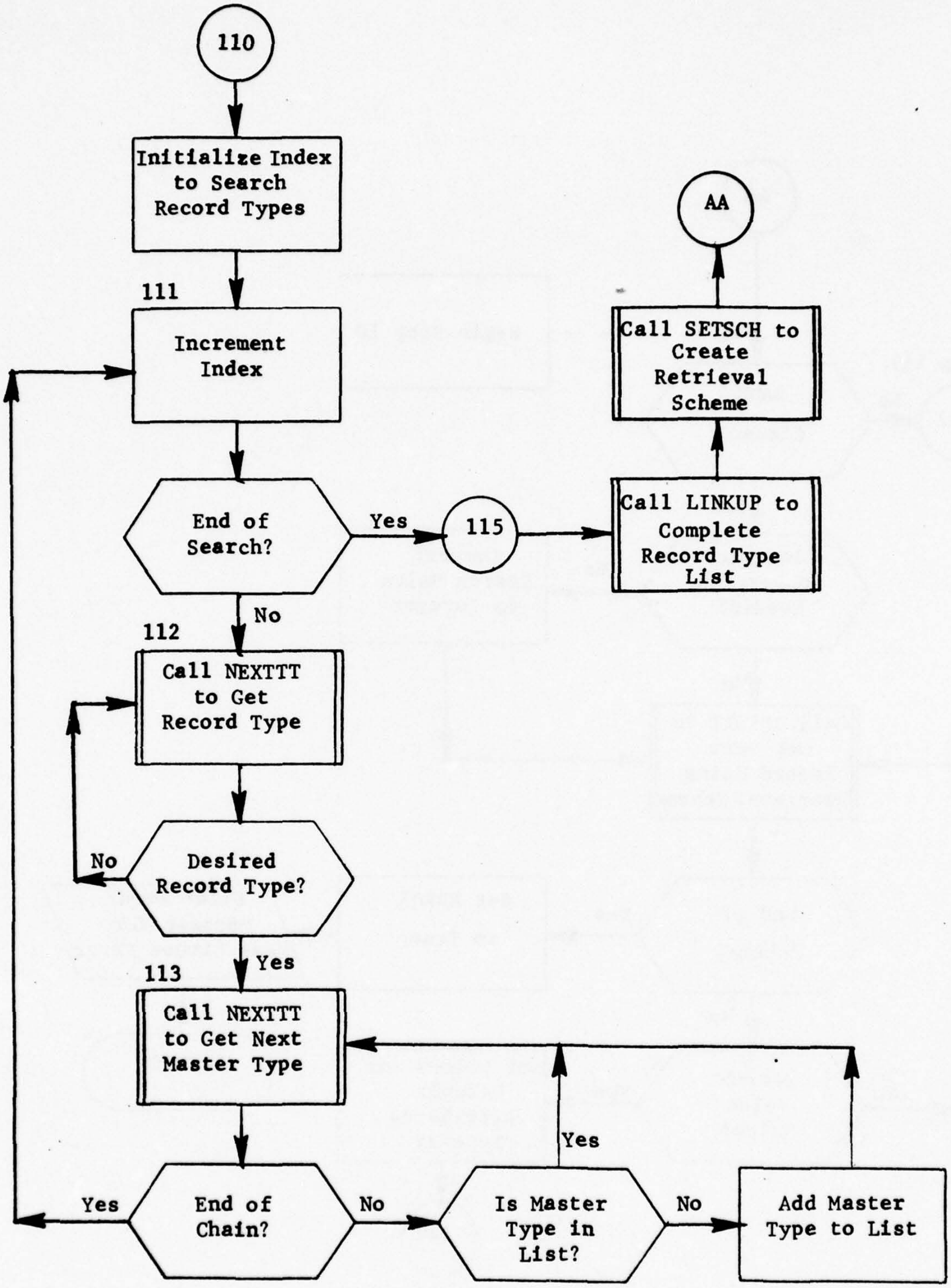



Figure 61. Subroutine CREAAT: Step 9

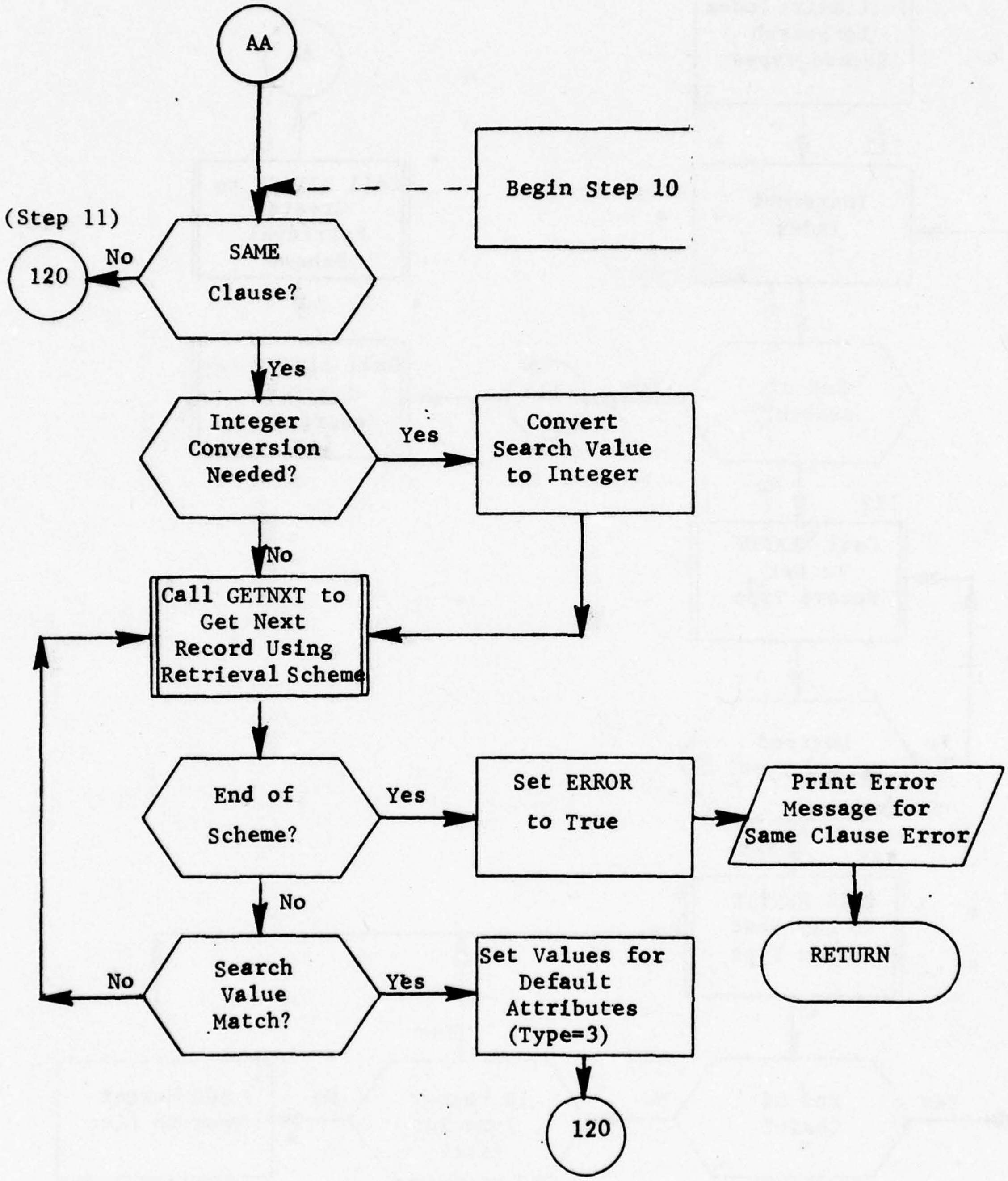



Figure 62. Subroutine CREAAT: Step 10

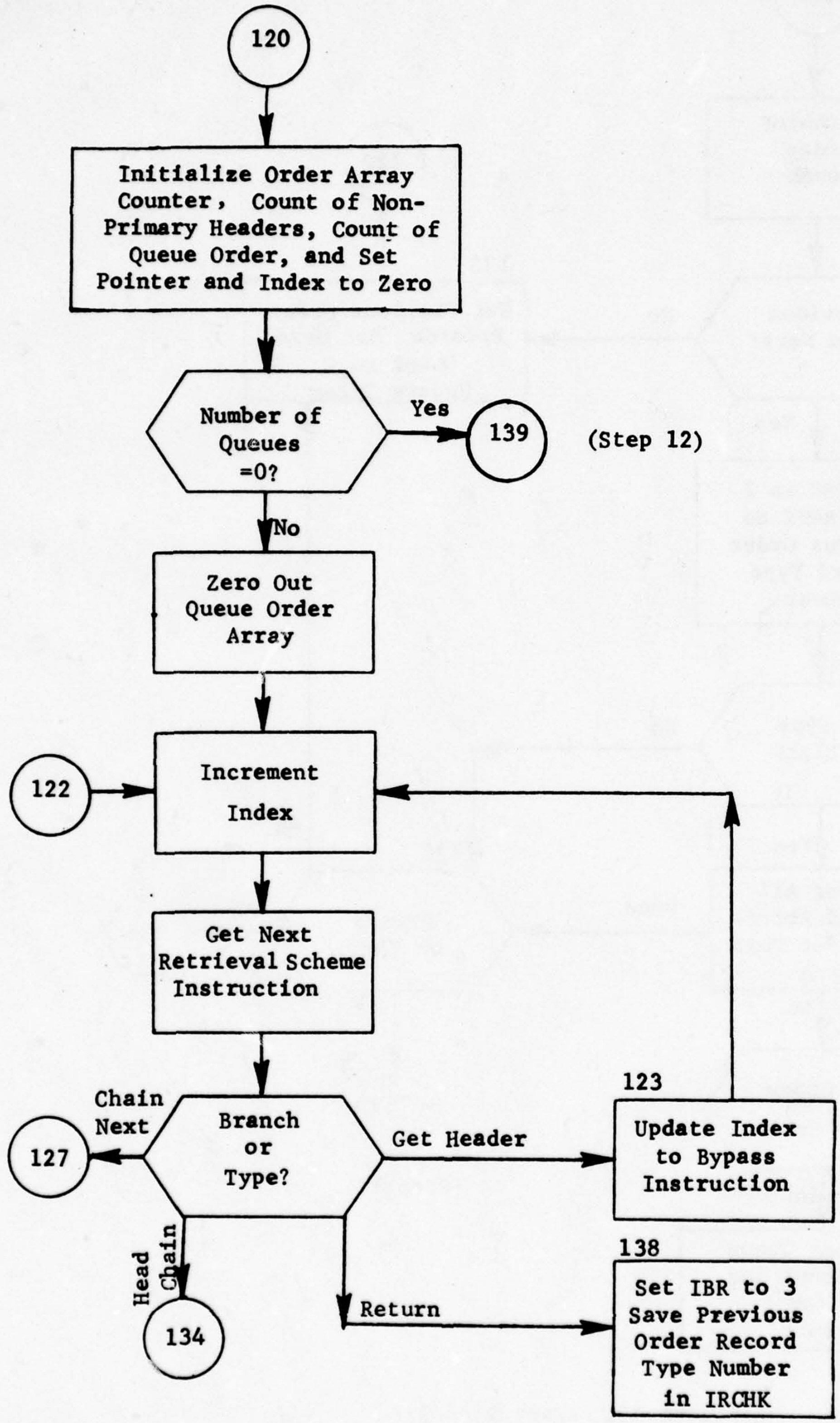



Figure 63. Subroutine CREAAT: Step 11 (Part 1 of 3)

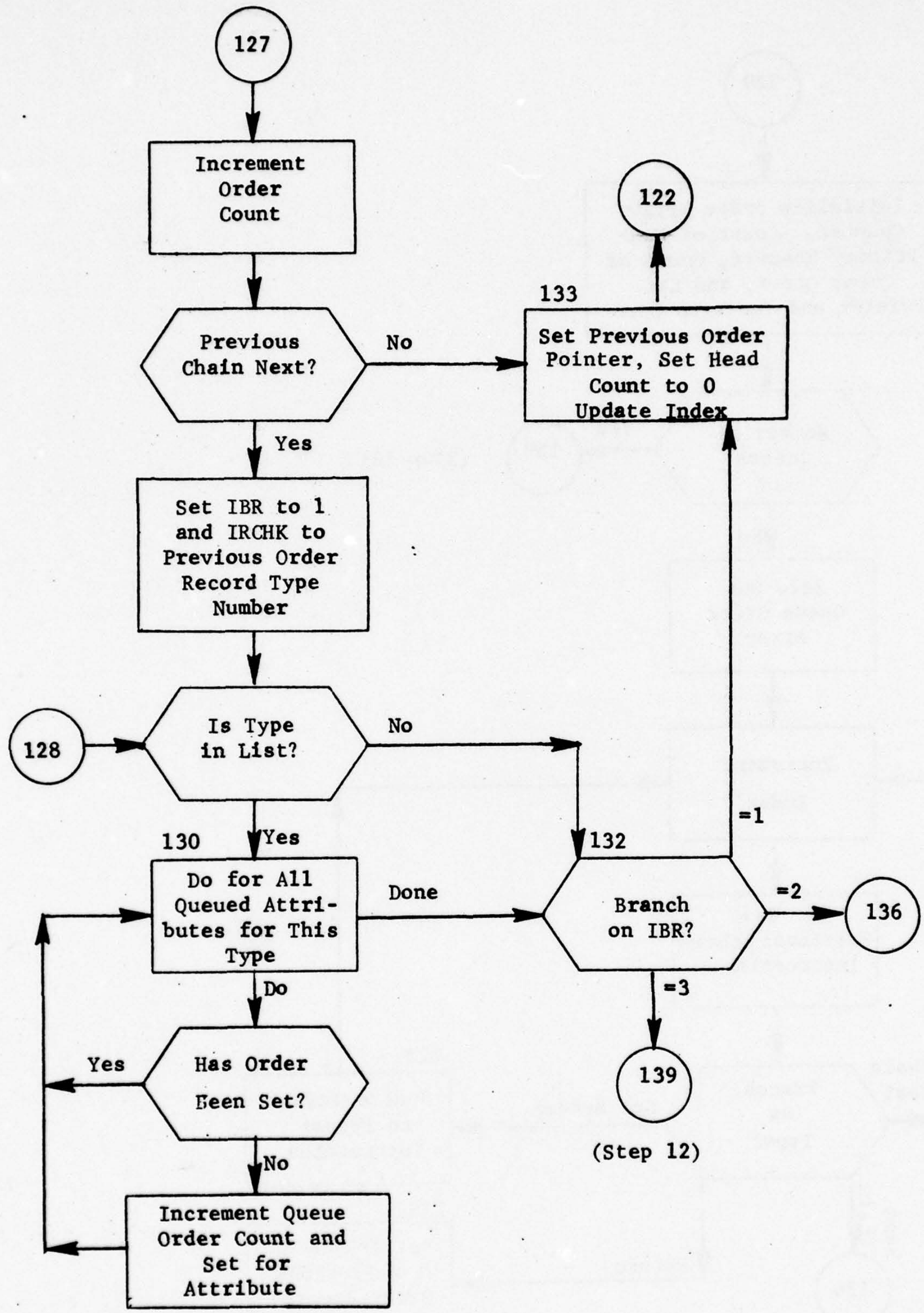



Figure 63. (Part 2 of 3)

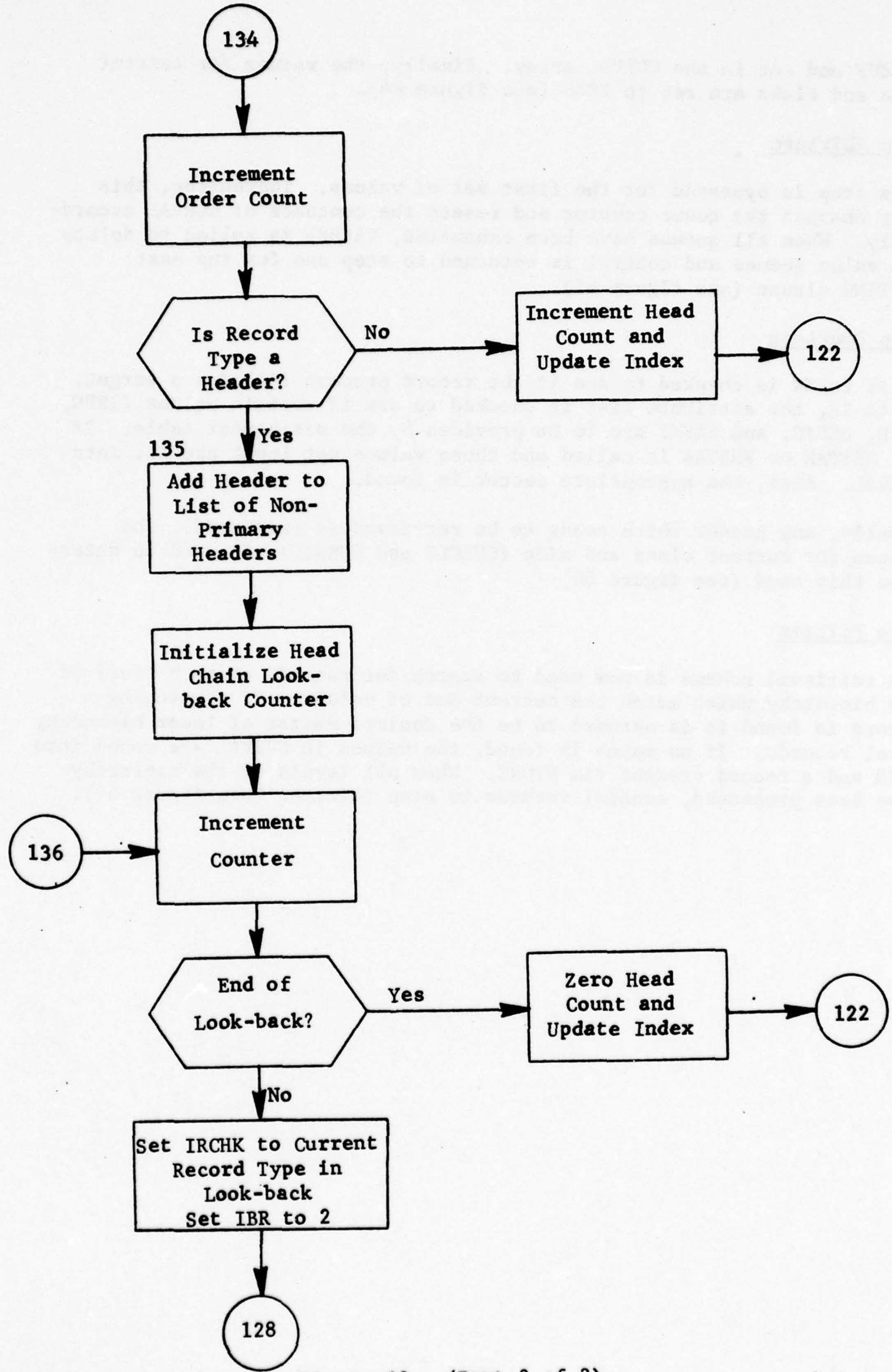



Figure 63. (Part 3 of 3)

VALBUF and set in the CURVAL array. Finally, the values for current side and class are set to ZEES (see figure 64).

Step Thirteen

This step is bypassed for the first set of values. Thereafter, this step changes the queue counter and resets the contents of CURVAL accordingly. When all queues have been exhausted, VALDEL is called to delete the value queues and control is returned to step one for the next SETTING clause (see figure 65).

Step Fourteen

First TGTSW is checked to see if the record process includes a target. If it is, the attribute list is checked to see if certain values (IREG, TYPE, DESIG, and TASK) are to be provided by the assignment table. If so, GETTAR or FNDTAR is called and those values not input are set into CURVAL. Also, the appropriate sector is found.

Finally, any header which needs to be retrieved is retrieved. The values for current class and side (CURCLS and CURSID) are used to determine this need (see figure 66)

Step Fifteen

The retrieval scheme is now used to search for records at each level of the hierarchy which match the current set of values. If a matching record is found it is assumed to be the desired master of lower hierarchy level records. If no match is found, the values in CURVAL are moved into MAIN and a record created via STORE. When all levels of the hierarchy have been processed, control returns to step thirteen (see figure 67).

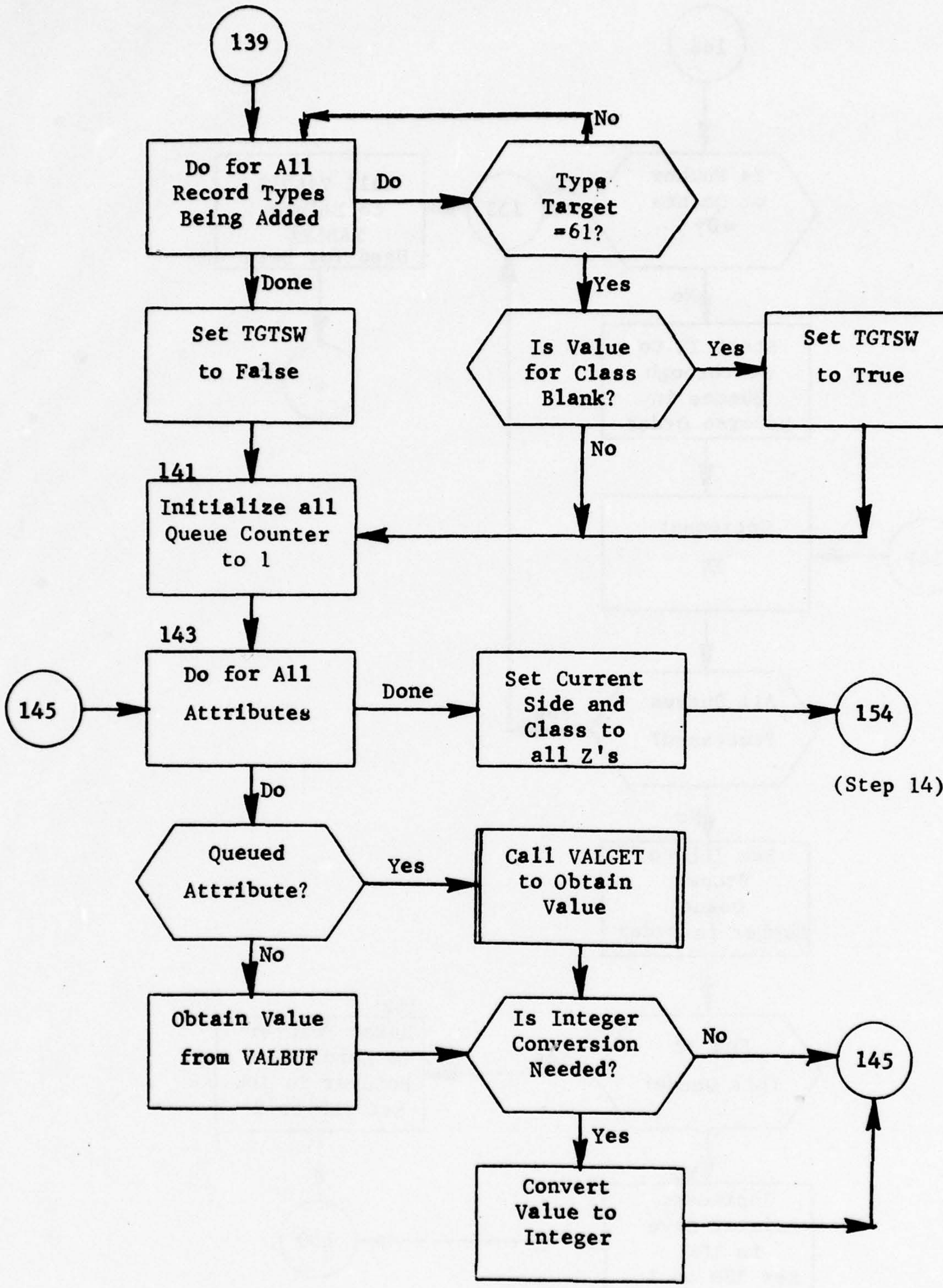



Figure 64. Subroutine CREAAT: Step 12

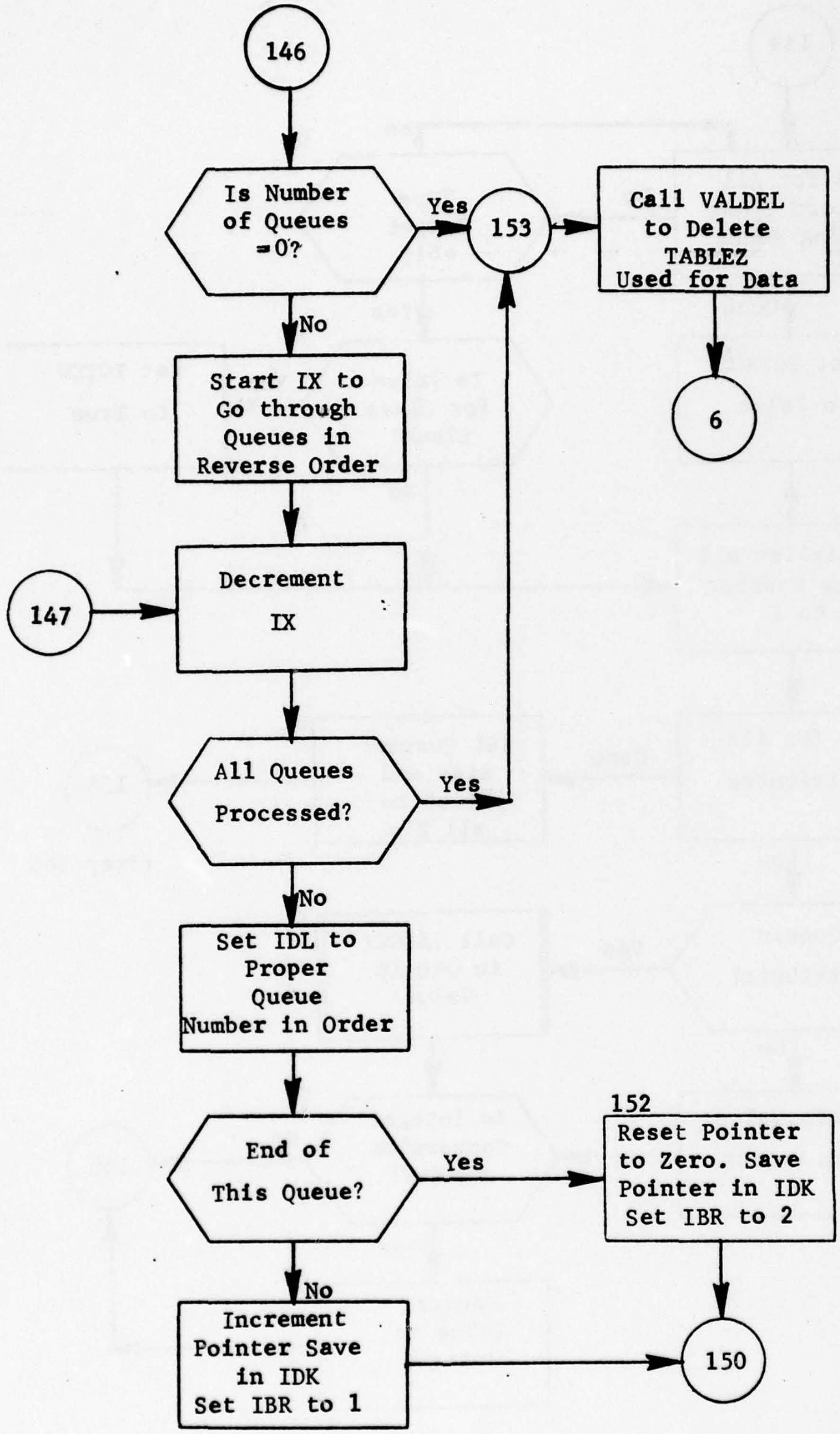



Figure 65. Subroutine CREAAT: Step 13 (Part 1 of 2)

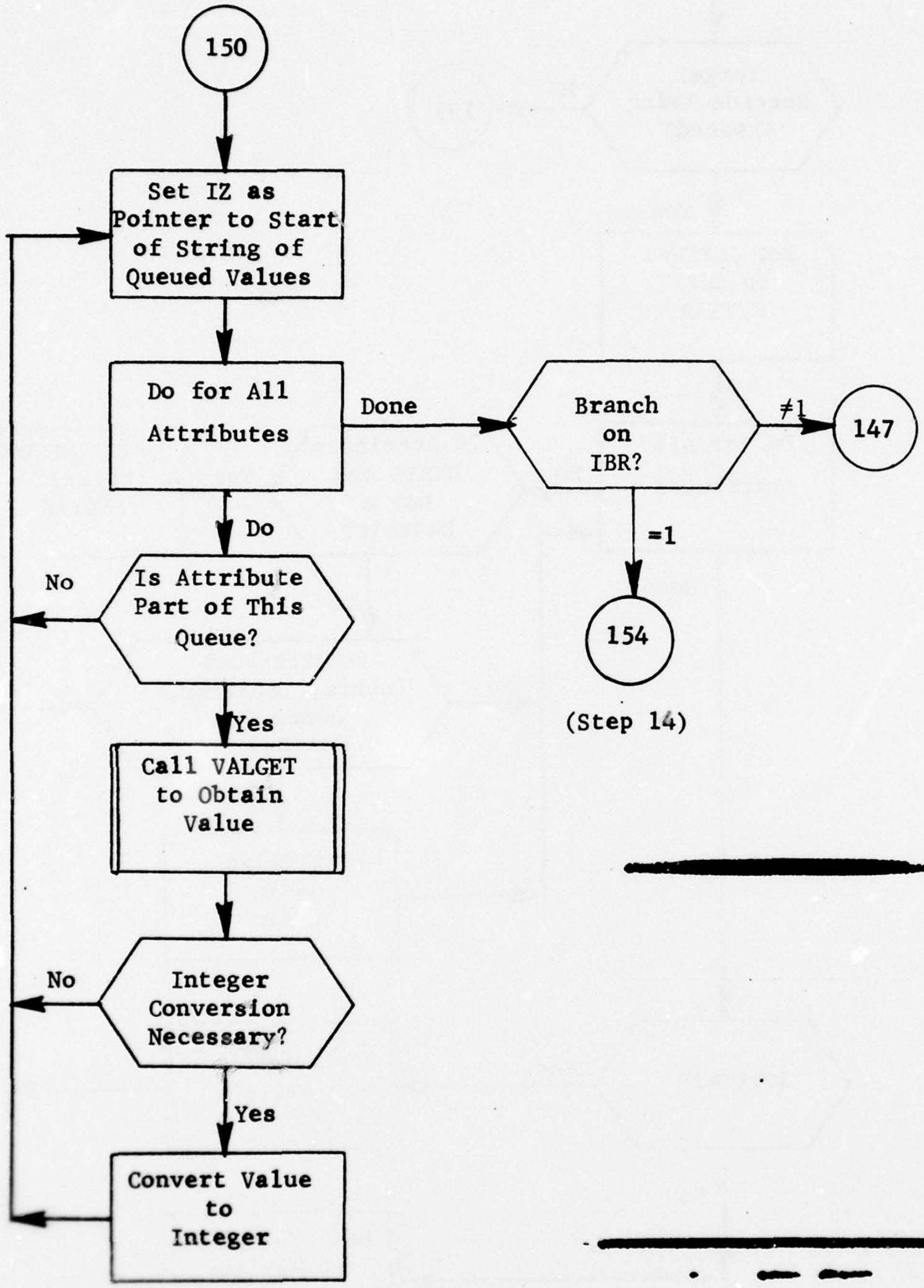



Figure 65. (Part 2 of 2)

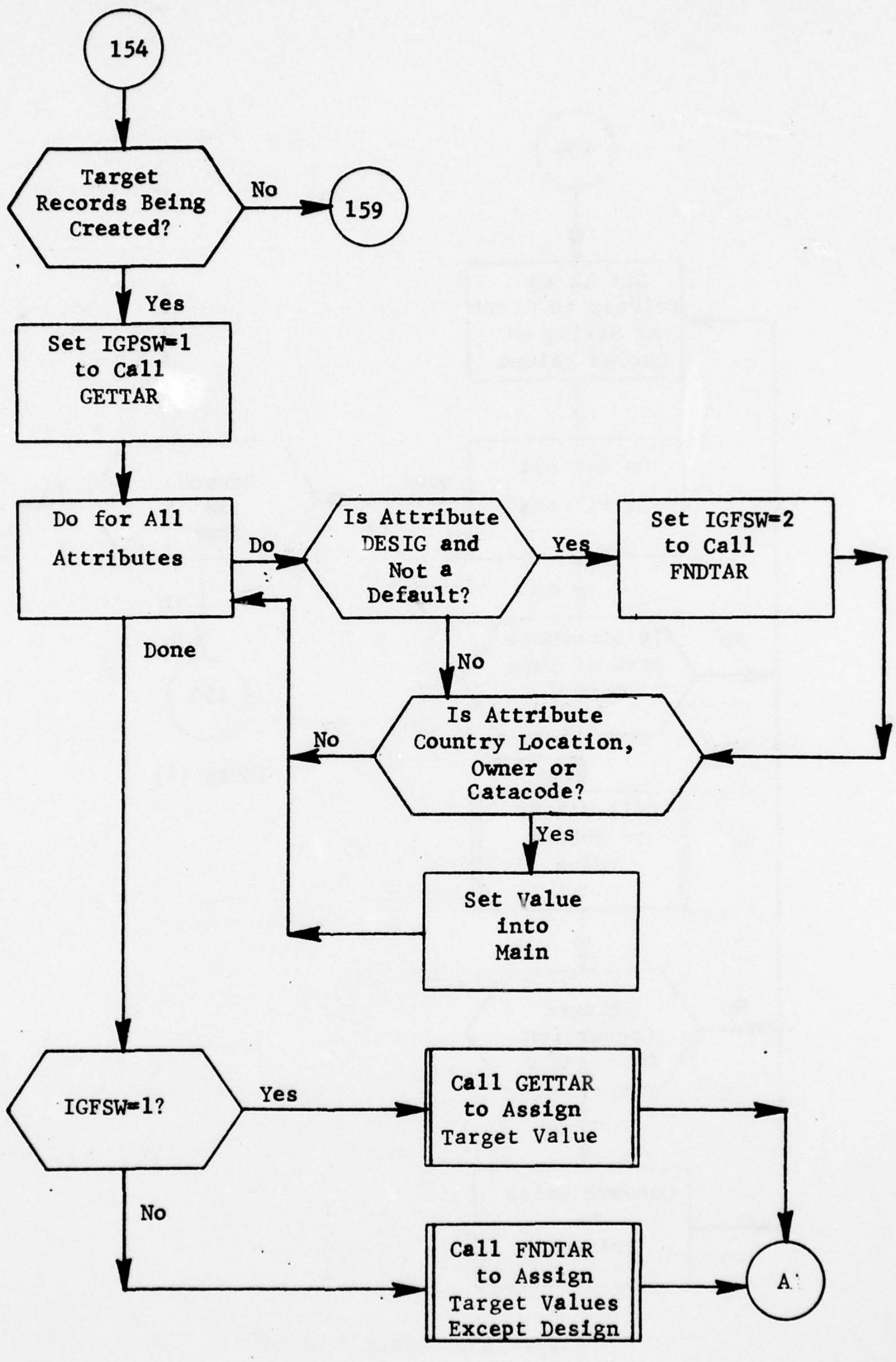



Figure 66. Subroutine CREAAT: Step 14 (Part 1 of 3)

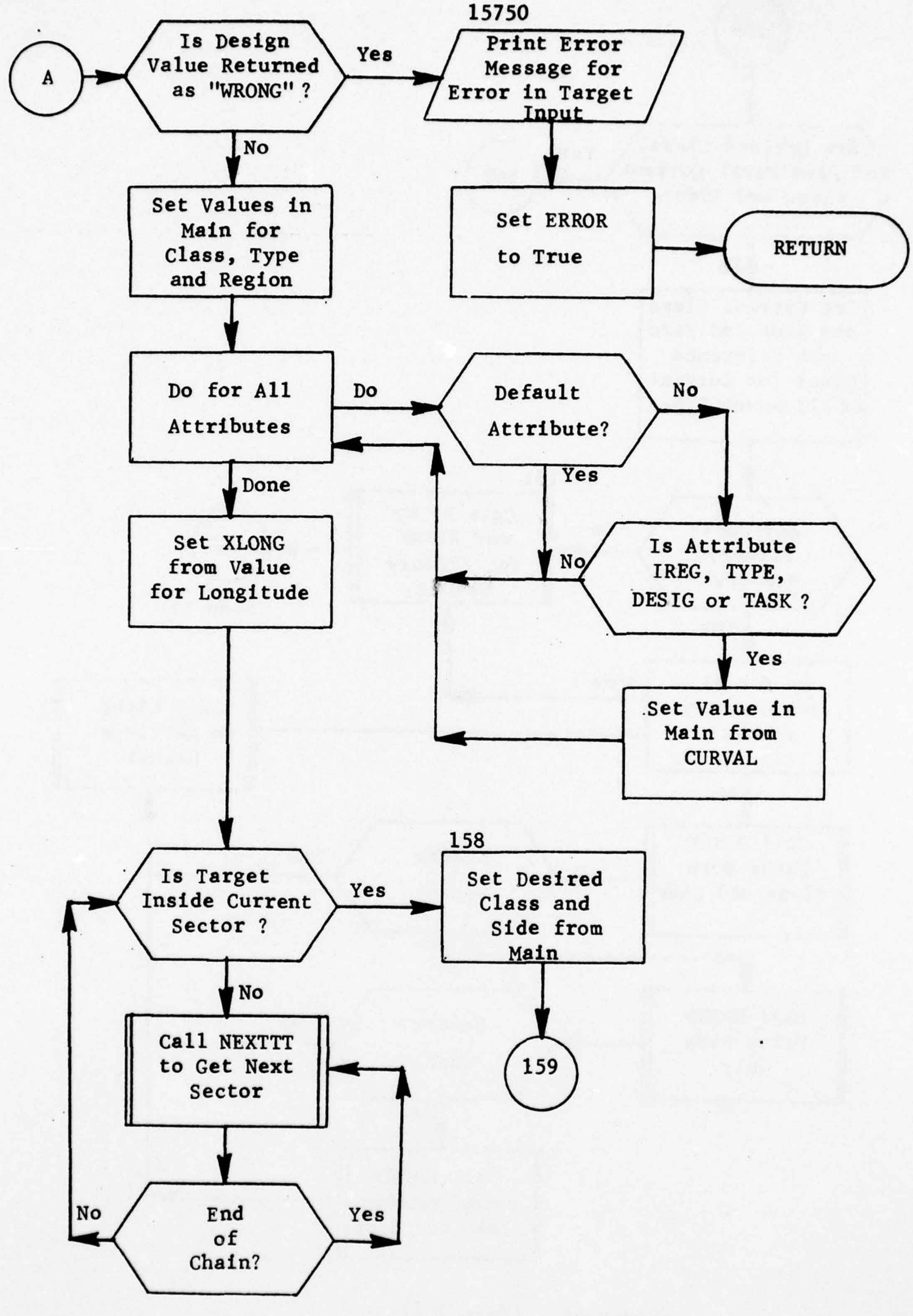



Figure 66. (Part 2 of 3)

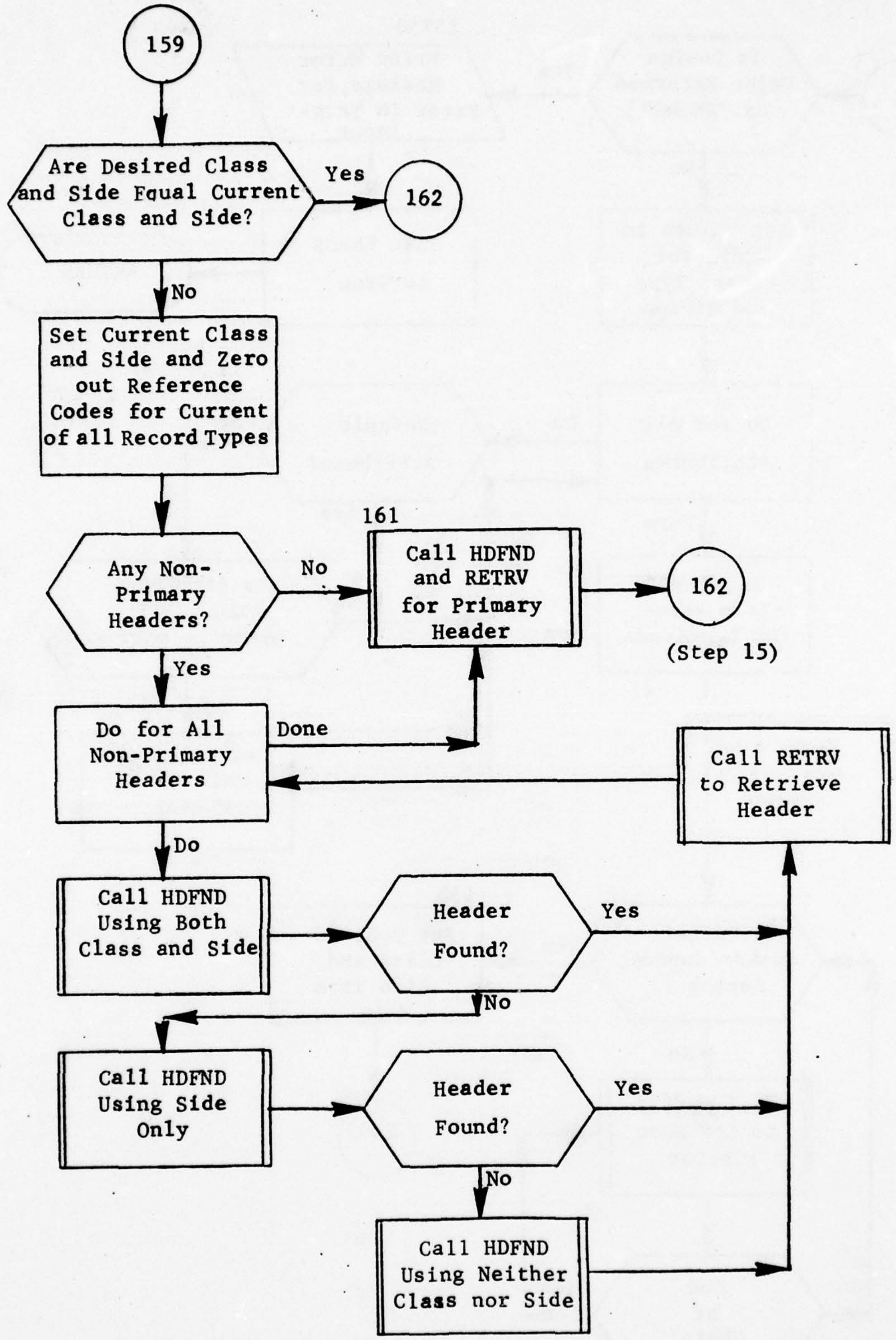



Figure 66. (Part 3 of 3)

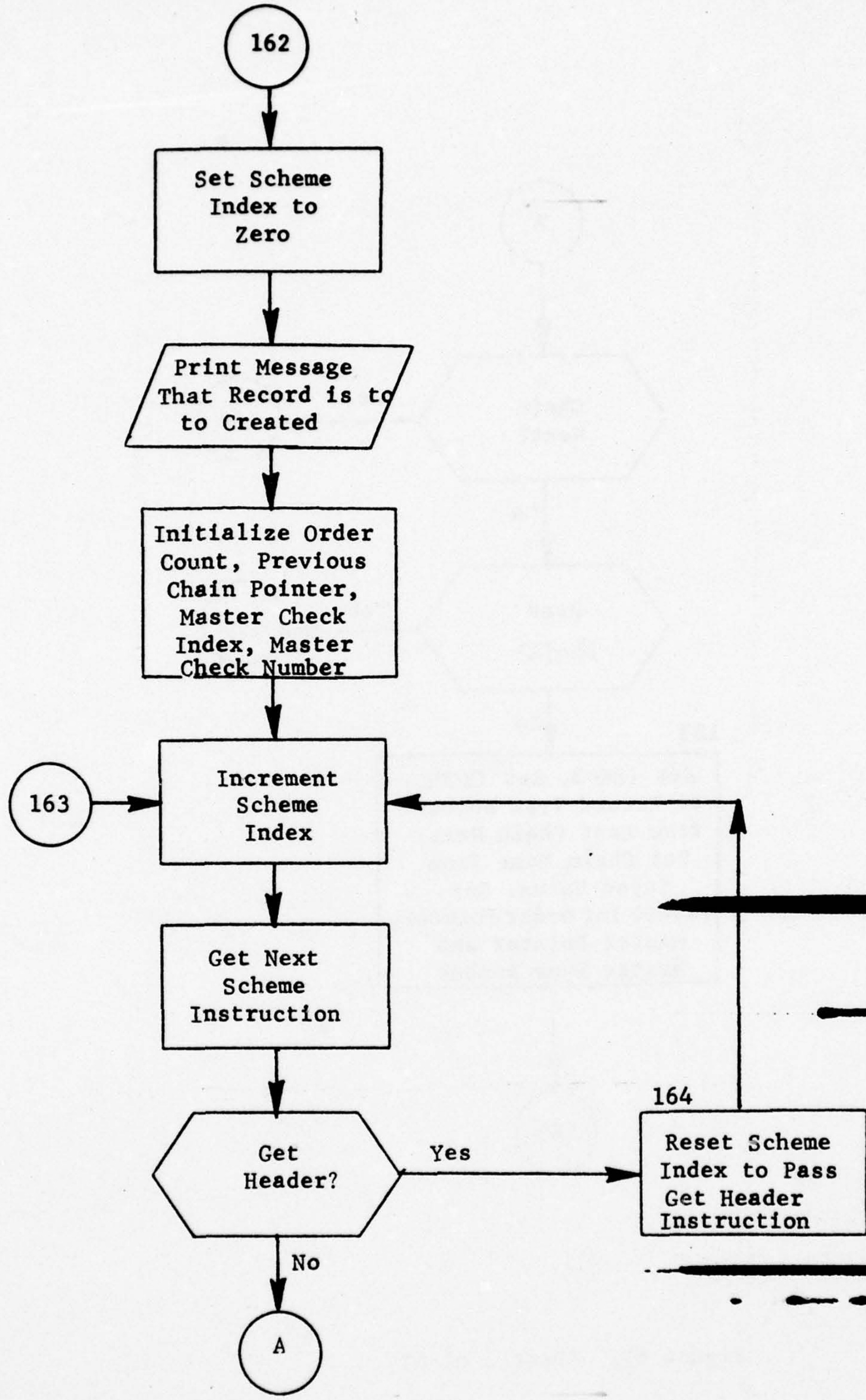



Figure 67. Subroutine CREAAT. Step 15 (Part 1 of 6)

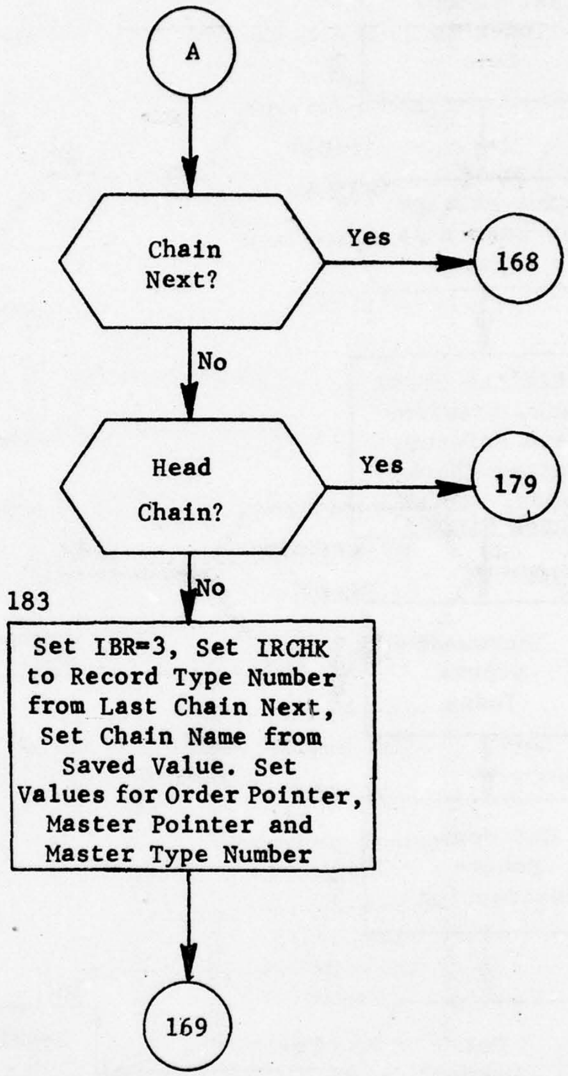



Figure 67. (Part 2 of 6)

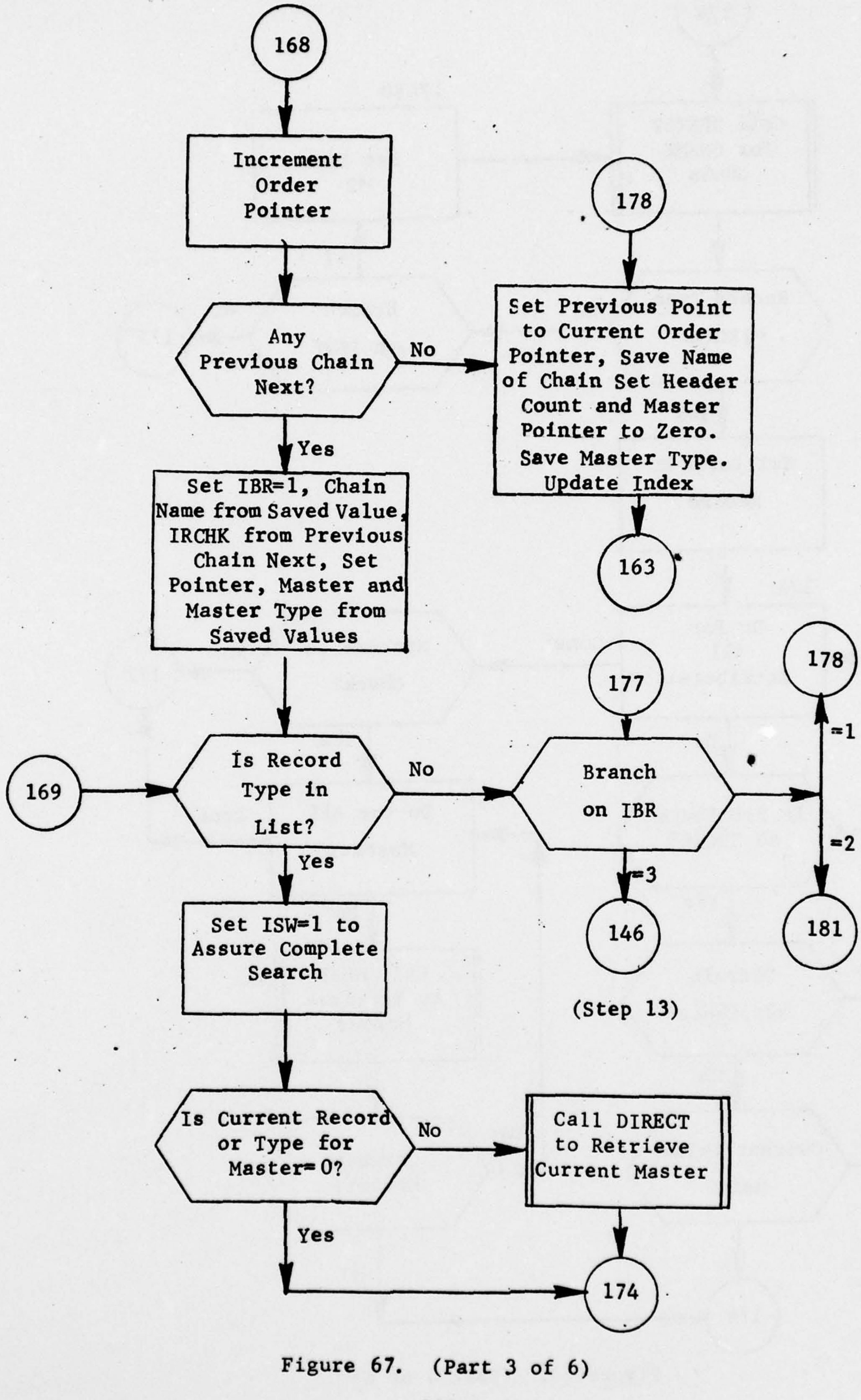



Figure 67. (Part 3 of 6)

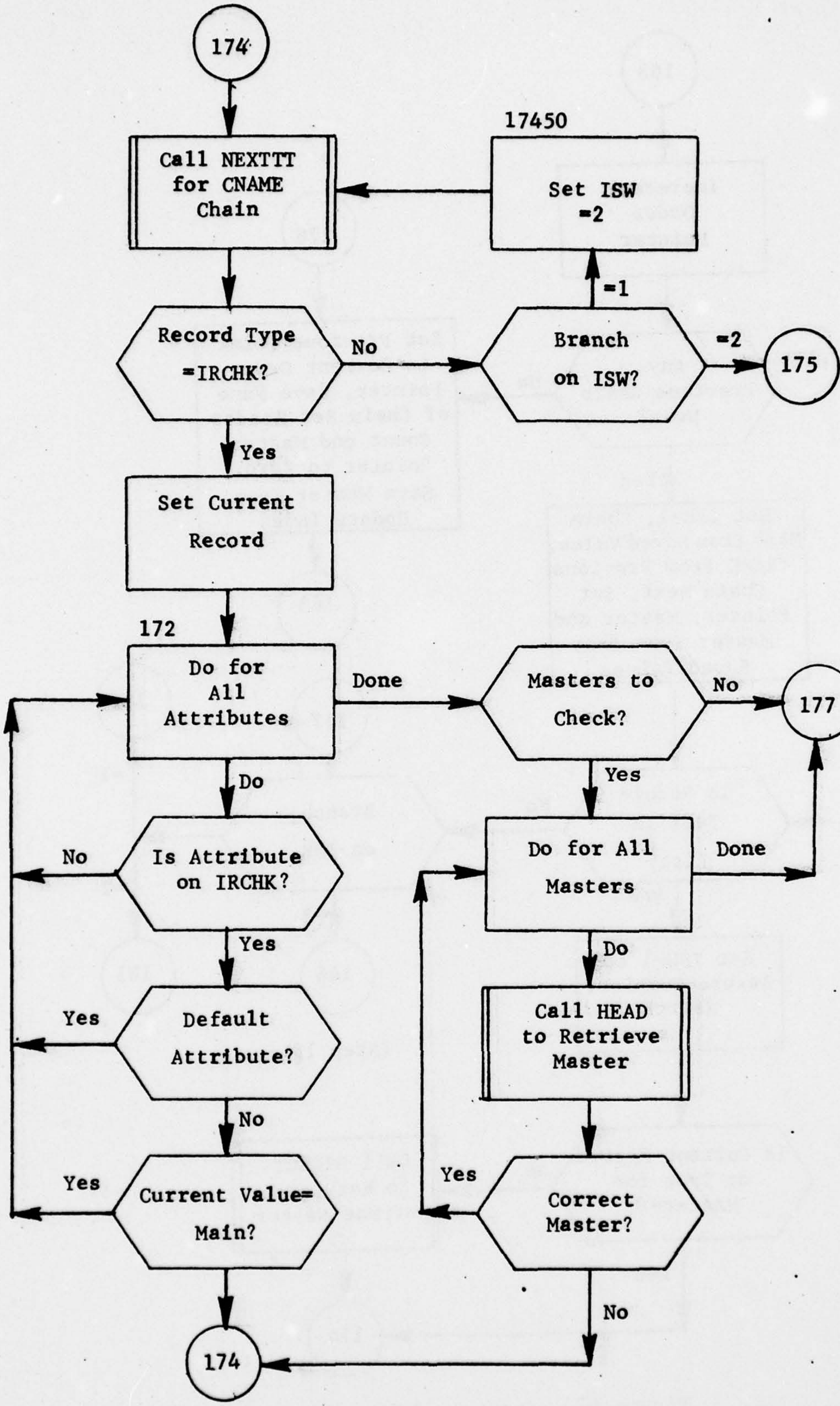



Figure 67. (Part 4 of 6)

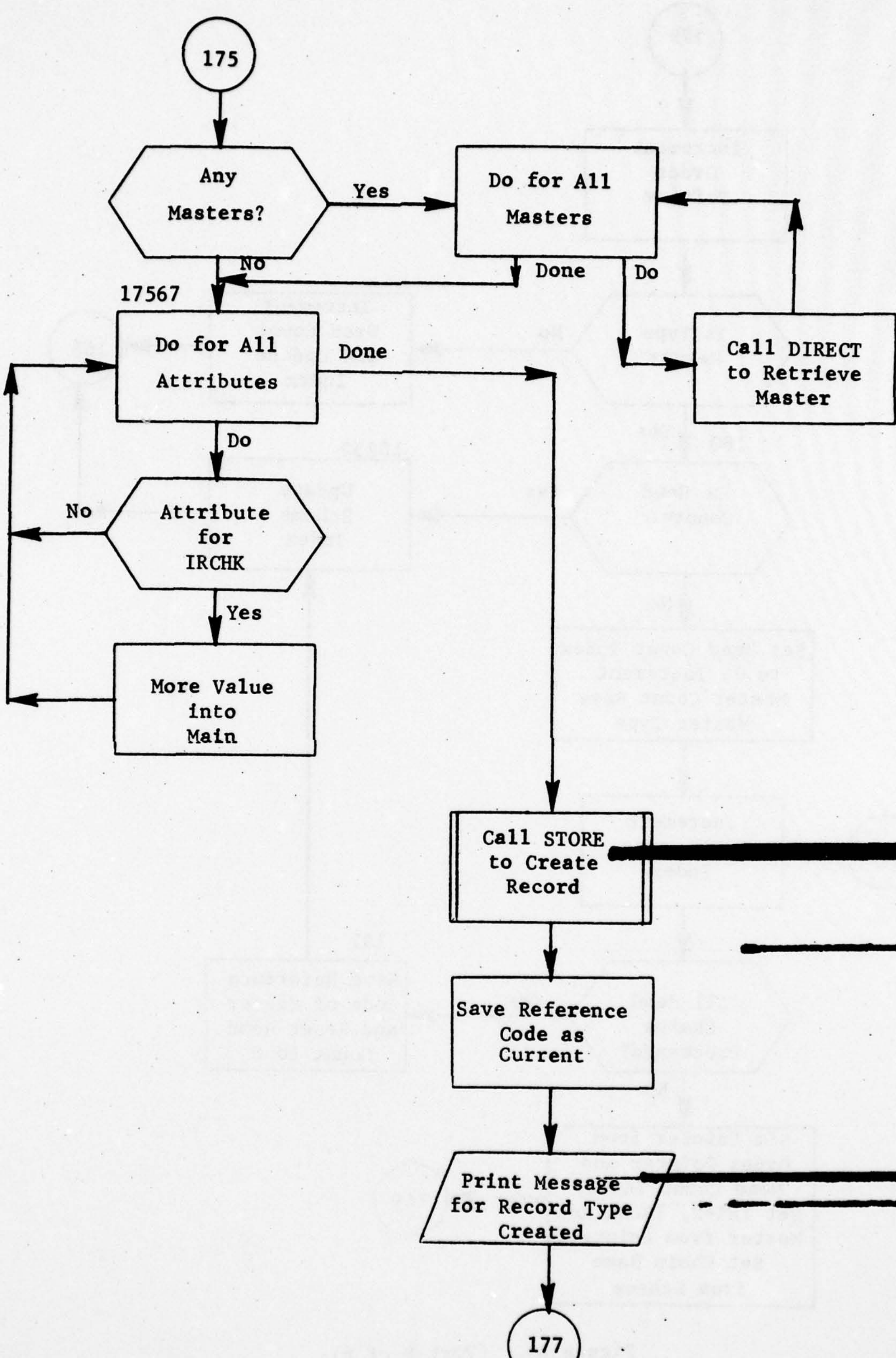



Figure 67. (Part 5 of 6)

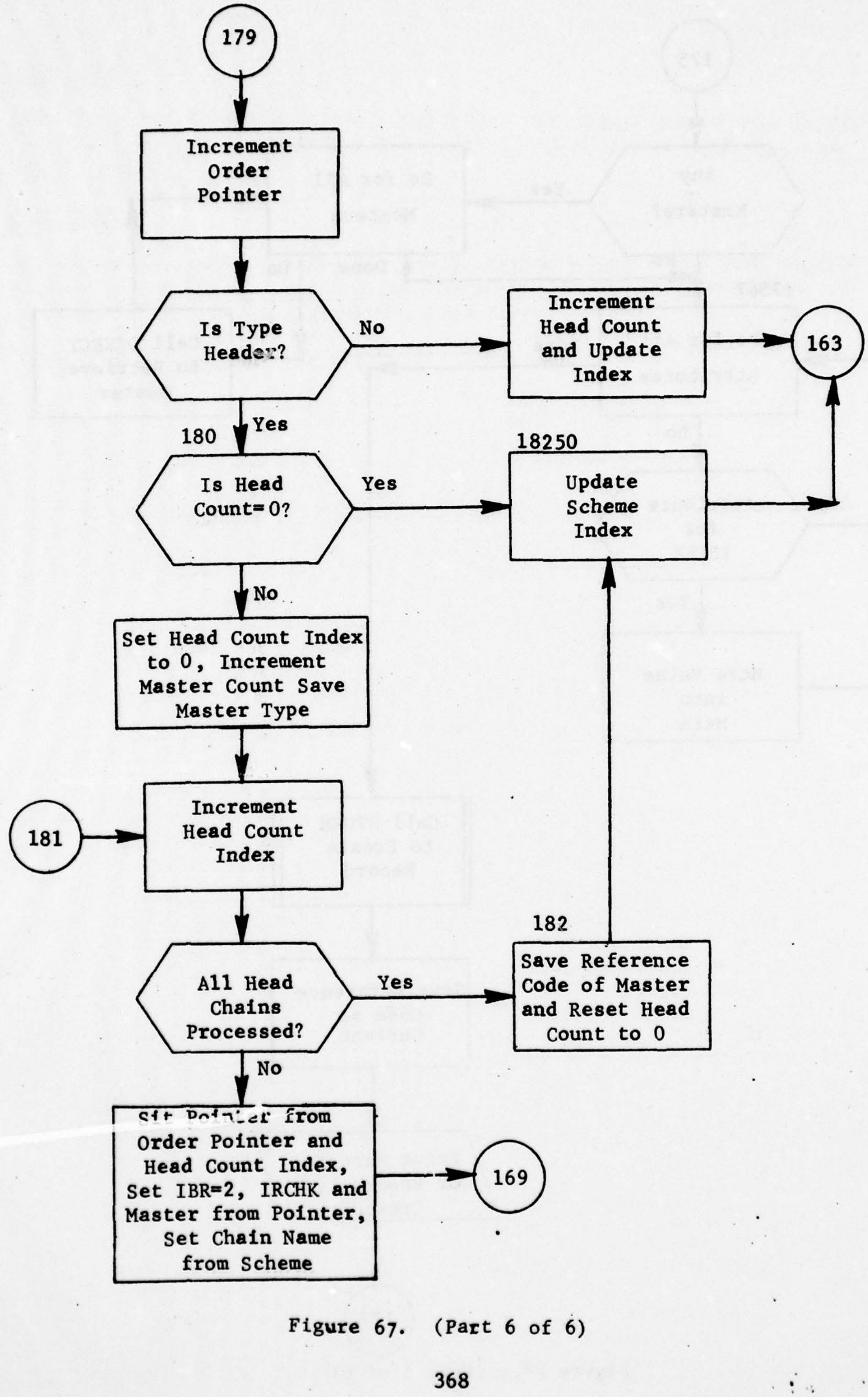



Figure 67. (Part 6 of 6)

4.10 Subroutine DELETE

PURPOSE: To delete records

ENTRY POINTS: DELETE

FORMAL PARAMETERS: None

COMMON BLOCKS: C10, C20, C30, OOPS, ORDER, PRWSP, SCHEME

SUBROUTINES CALLED: DLETE, GETNXT, HDFND, HEAD, INSGET, LINKUP, NEXTTT, OFVAL, PRIMHD, SETSCH, UNCODE, VALFND, XWHERE

CALLED BY: ENTMOD (DATA)

Method:

The DELETE verb process can be broken into five steps which are executed in sequence.

Step One

The WHERE clause is scanned. In the process OF phrases and LIKE strings are resolved immediately using VALFND and the values stored by OFVAL. Attributes which are found are placed in a list (ATNUMB) and if either the CLASS or SIDE attributes are encountered the values given are saved to assist in the retrieval scheme construction process.

Step Two

The list of attributes (ATNUMB) is now resolved via the ATRIB chain to create a list of record types (RTLIST). A single defined attribute record type is added to the list. A control attribute has its controlled record added to the list. The record types which contain multiply defined attributes are saved in a separate list (MLTLST).

Step Three

The primary header is now determined in one of two ways. If a value was given for CLASS, HDFND is called for the primary header. Otherwise the highest numbered record is determined and PRIMHD is called. Now the chains of which the primary header is the master are searched looking for matches among the list of multiple attribute record types. Any matches are added to the record type list. Finally LINKUP and SETSCH are called to build the retrieval scheme.

Step Four

The retrieval scheme is used to determine the lowest record type in the hierarchy to be retrieved and this type is noted as the type to be deleted (DNAME).

Step Five

Now GETNXT is called to execute the retrieval scheme. For each record retrieved, XWHERE is called to determine if the record is desired. If so DLETE is called for the DNAME record type.

Subroutine DELETE is illustrated in figure 68.

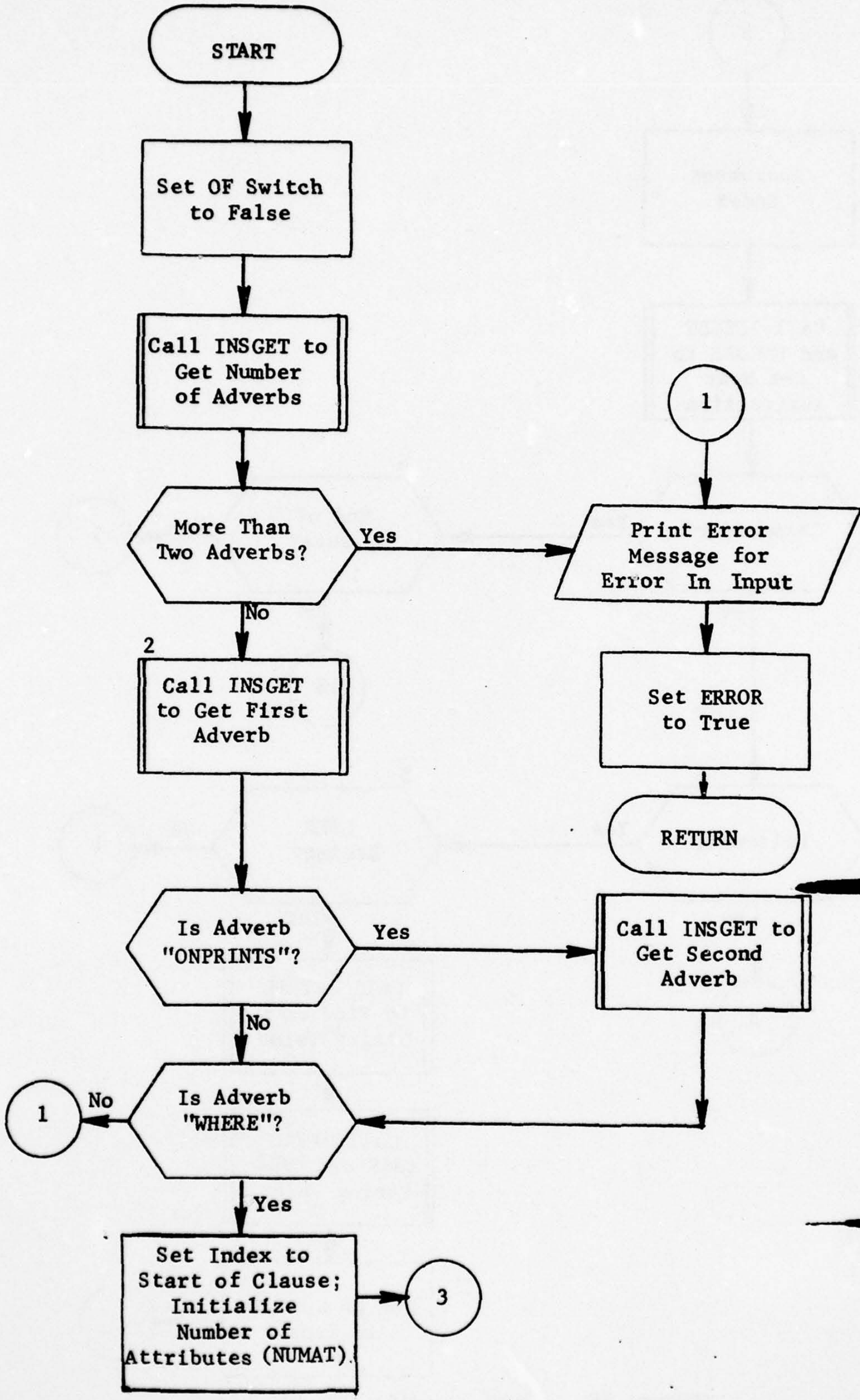



Figure 68. Subroutine DELETE (Part 1 of 12)

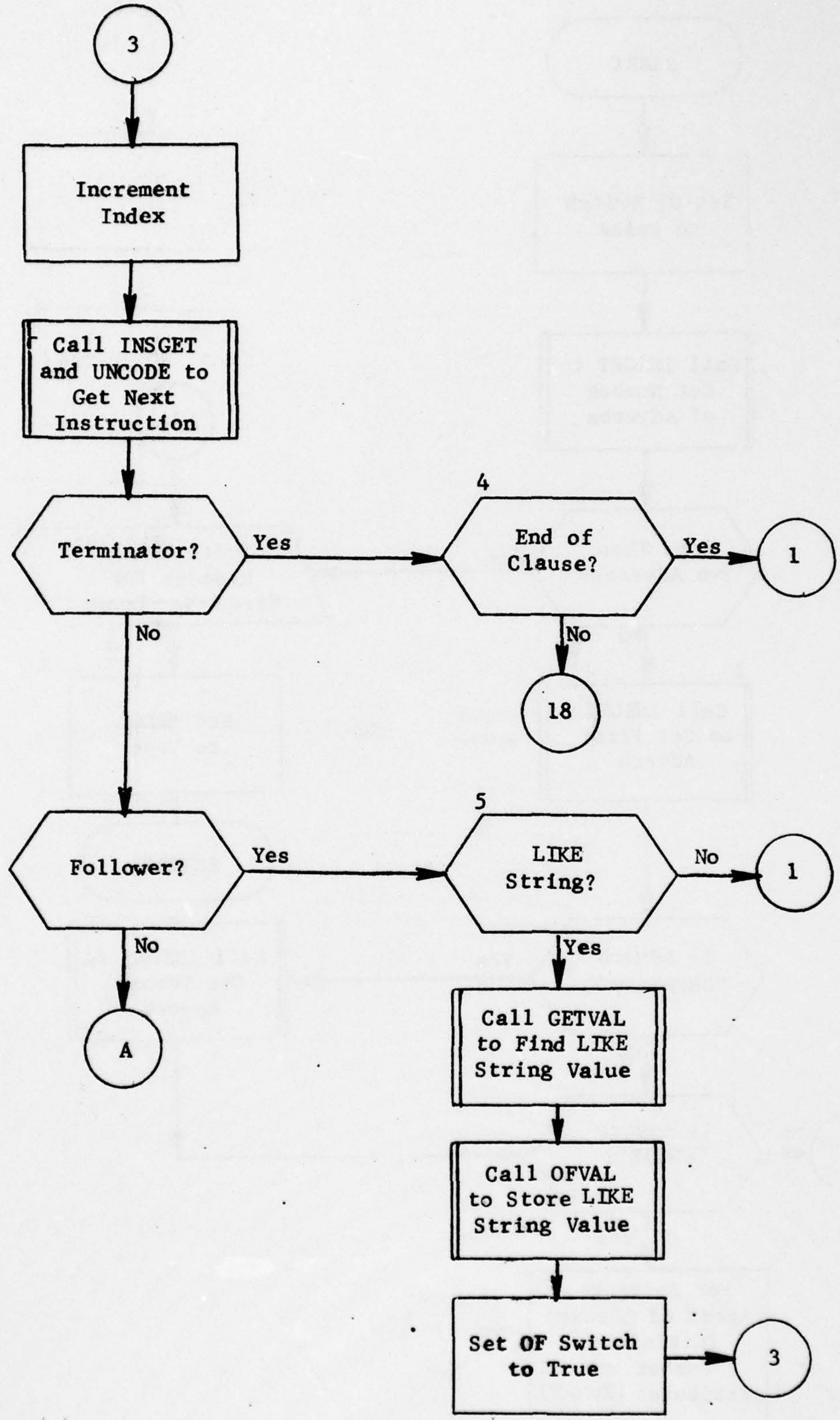



Figure 68. (Part 2 of 12)

AD-A054 377 COMMAND AND CONTROL TECHNICAL CENTER WASHINGTON D C F/G 15/7
THE CCTC QUICK-REACTING GENERAL WAR GAMING SYSTEM. (QUICK). PRO--ETC(U)
JUN 77 D J SANDERS, P F MAYKRANTZ, J M HERRIN
UNCLASSIFIED CCTC-CSM-MM-9-77-V1-PT-1 SBIE-AD-E100 051 NL

5 OF 5
AD
A054377

END
DATE
FILMED
6-78
DDC

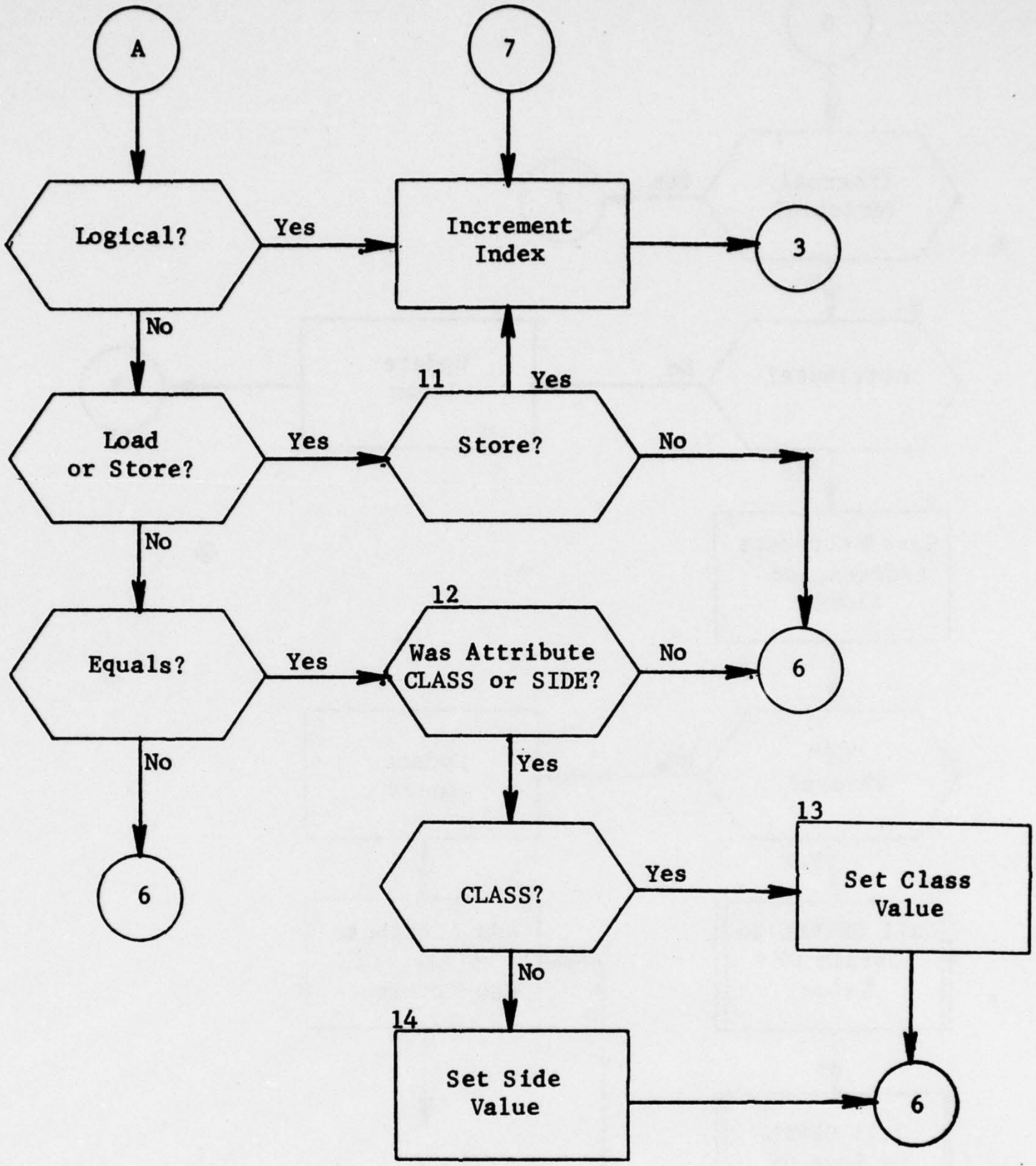



Figure 68. (Part 3 of 12)

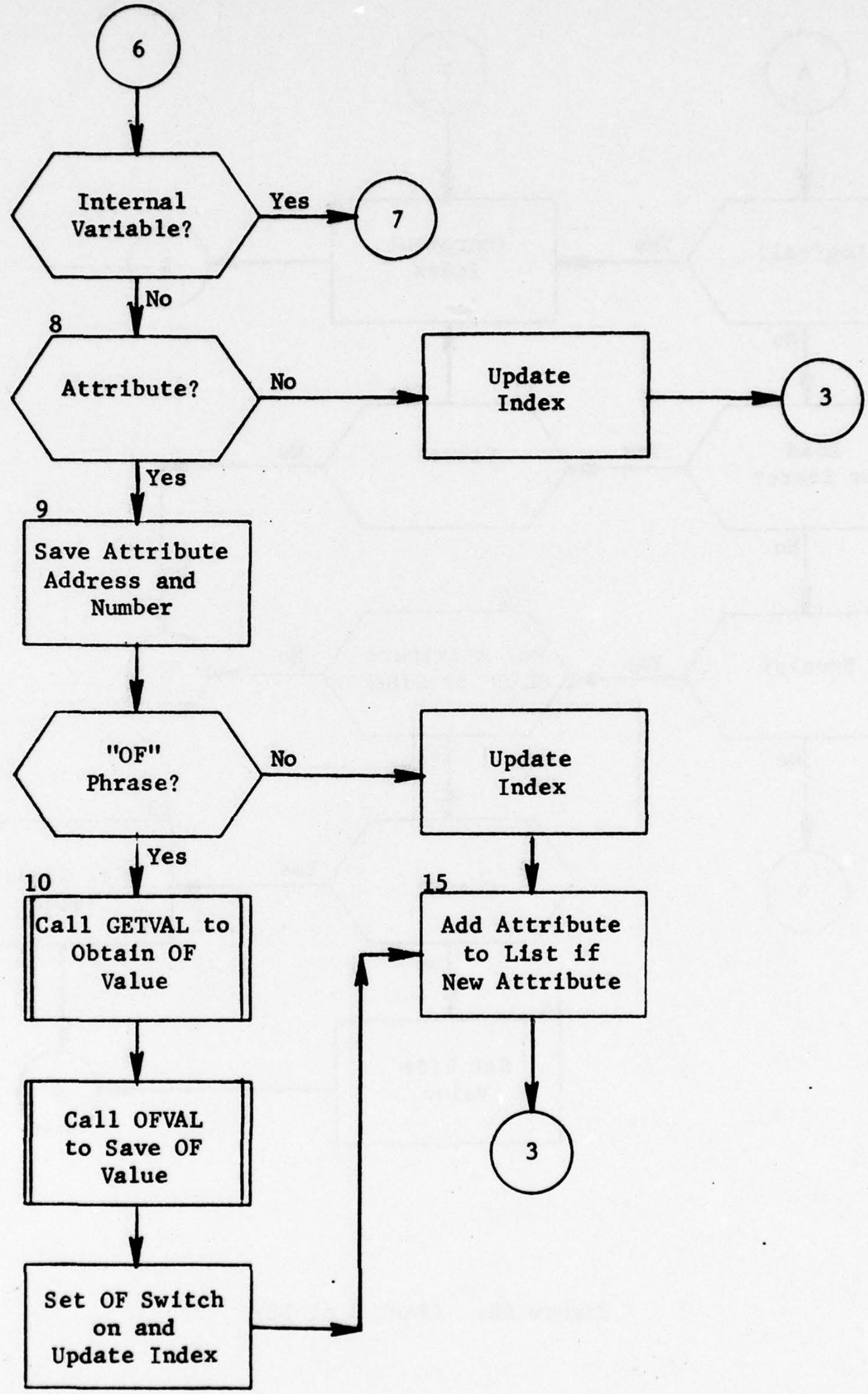



Figure 68. (Part 4 of 12)

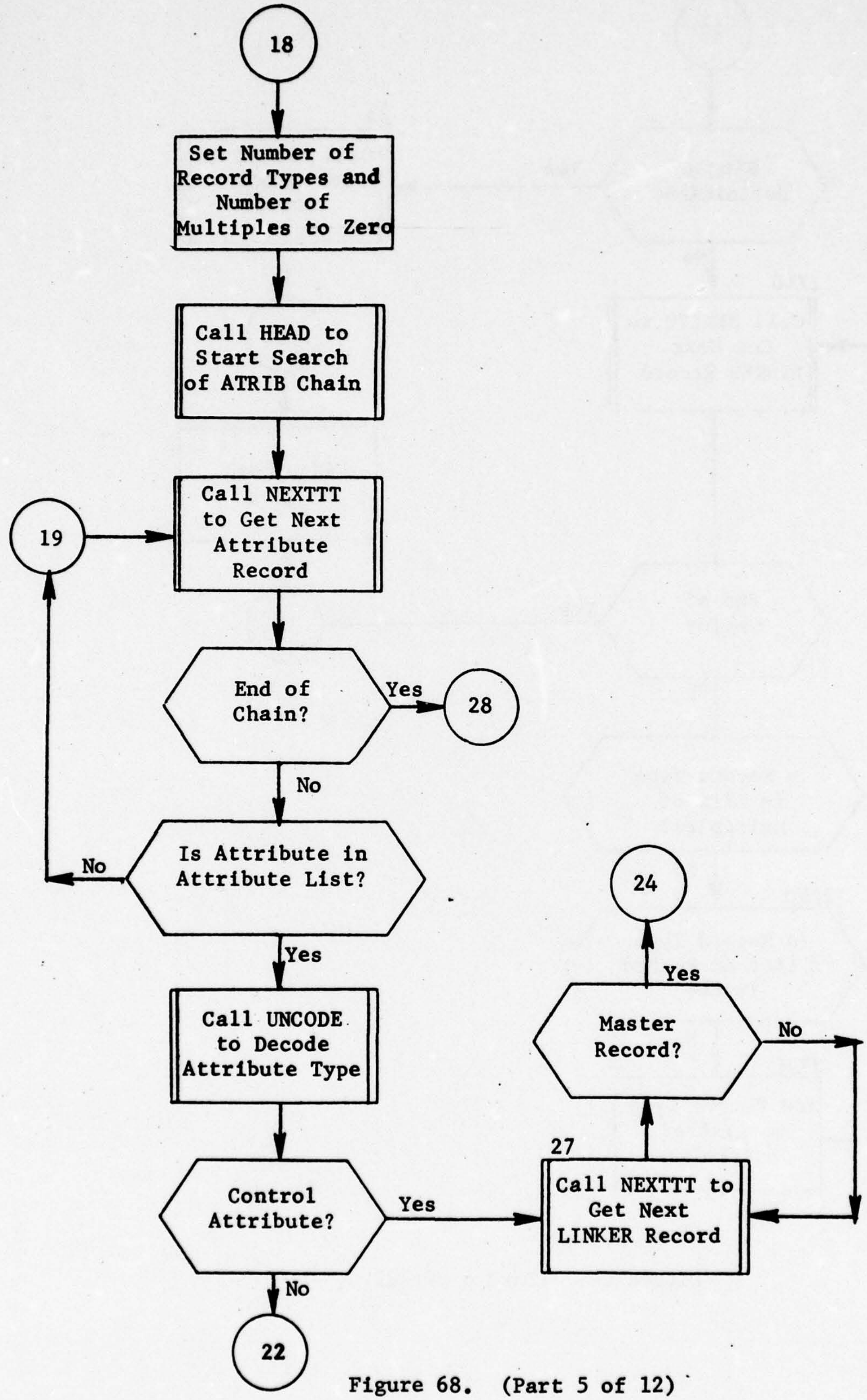



Figure 68. (Part 5 of 12)

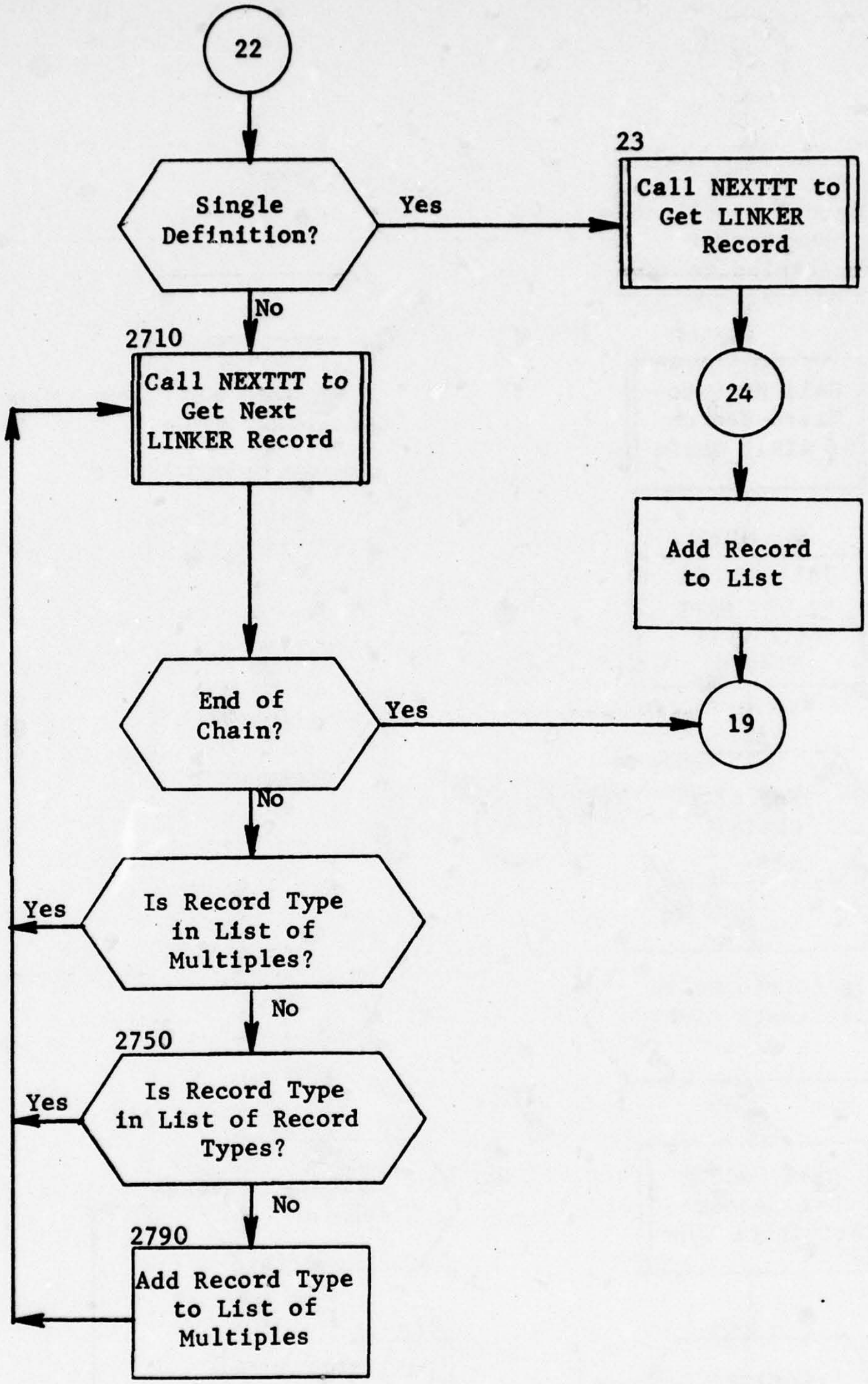



Figure 68. (Part 6 of 12)

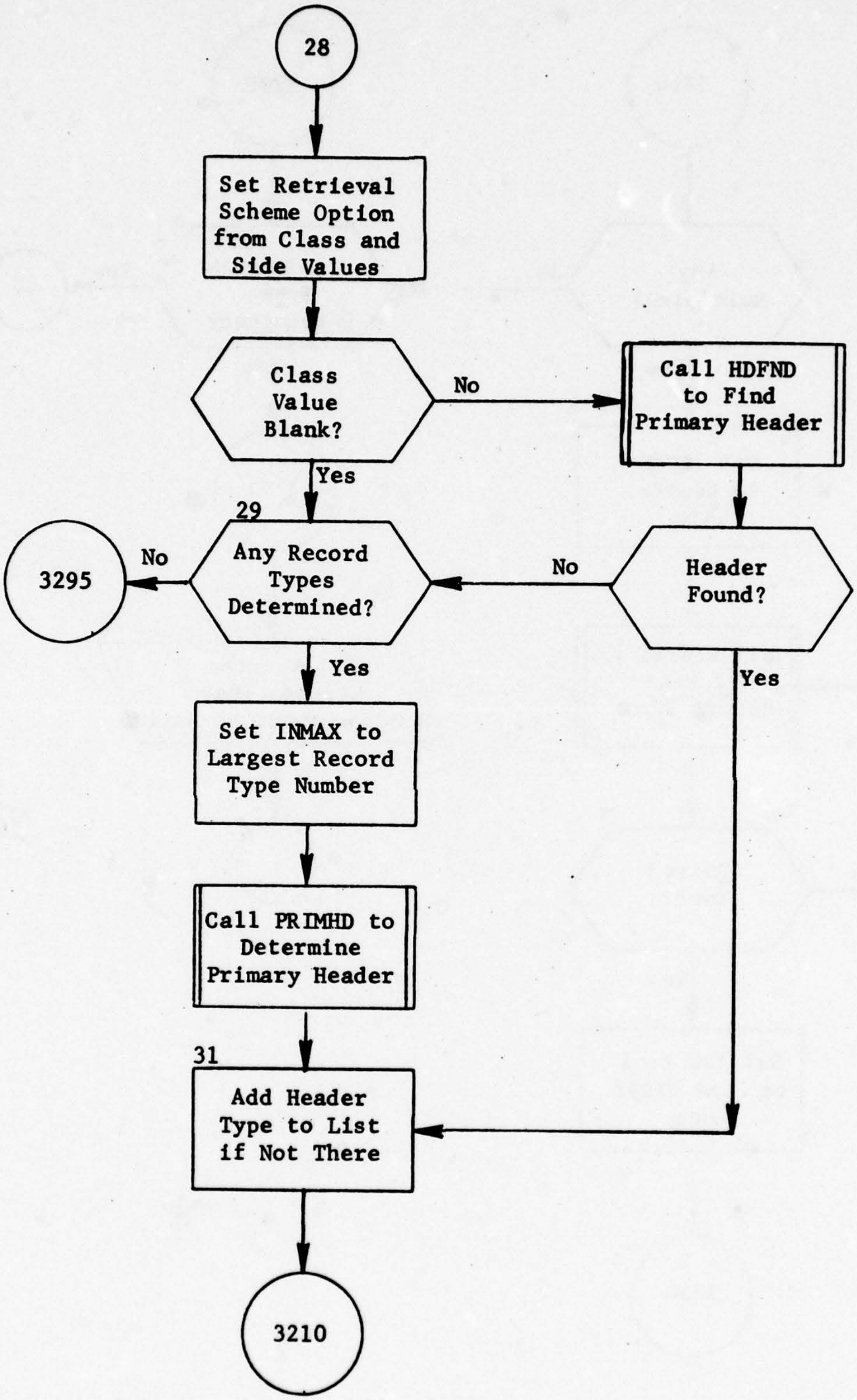



Figure 68. (Part 7 of 12)

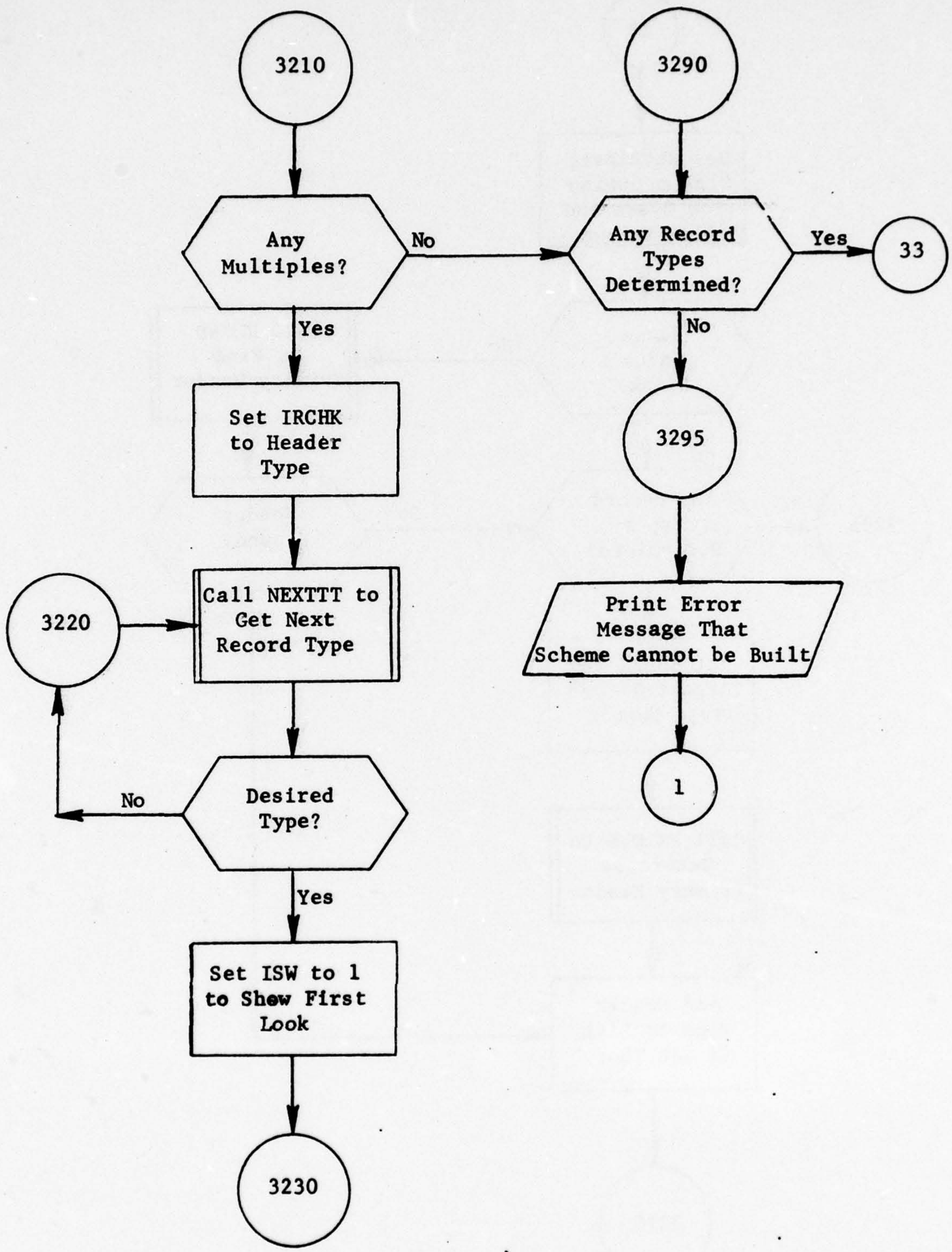



Figure 68. (Part 8 of 12)

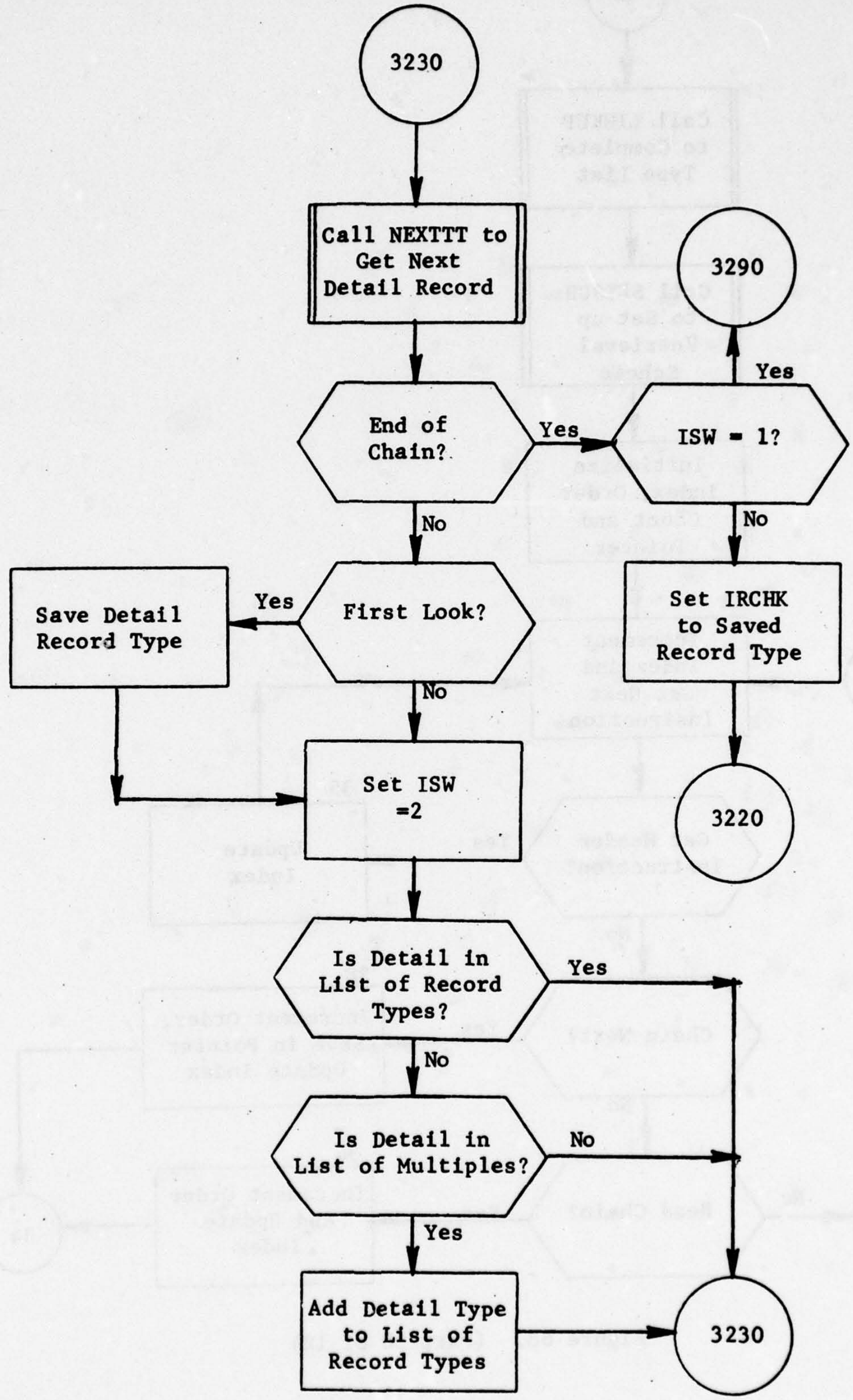



Figure 68. (Part 9 of 12)

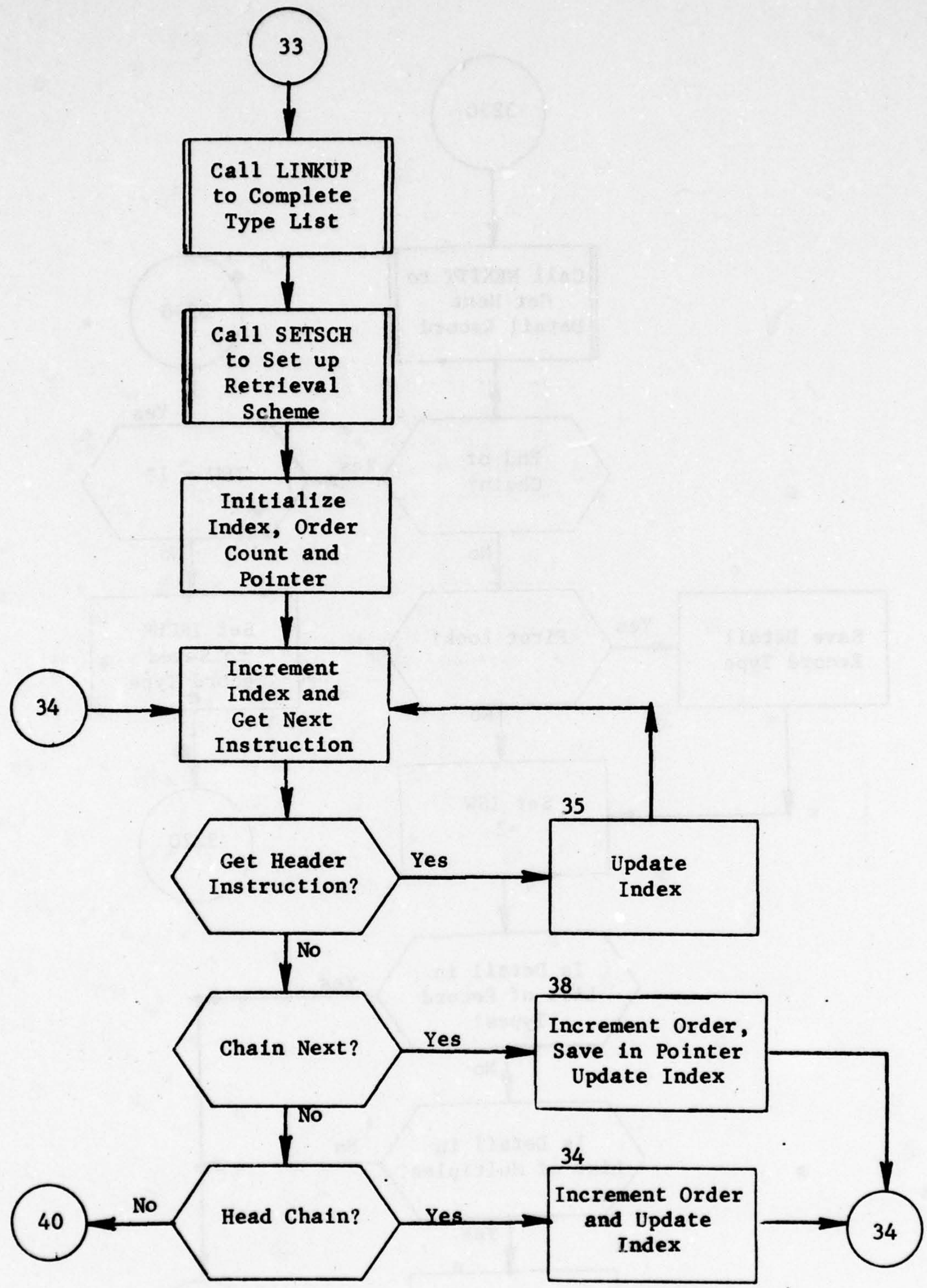



Figure 68. (Part 10 of 12)

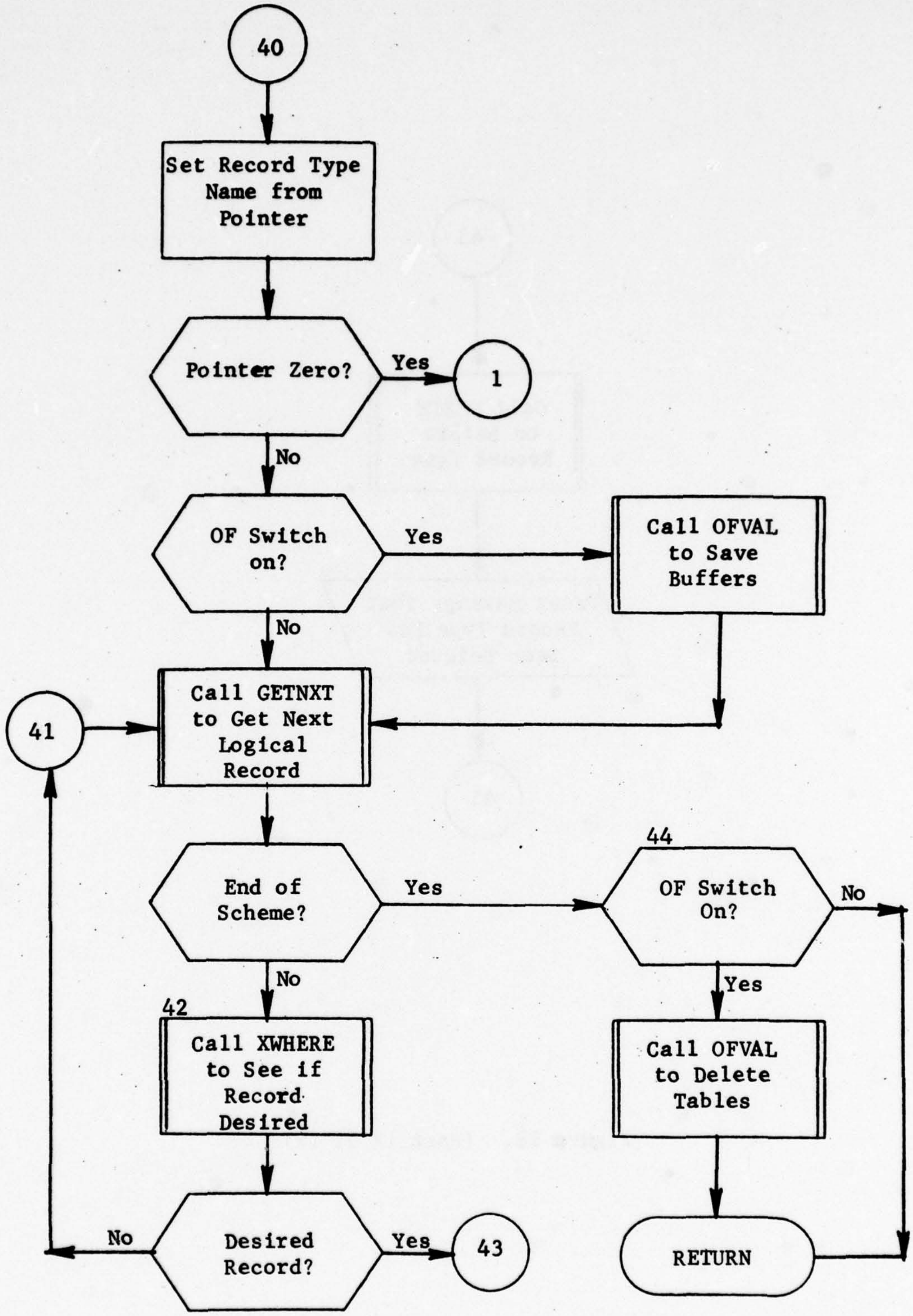



Figure 68. (Part 11 of 12)

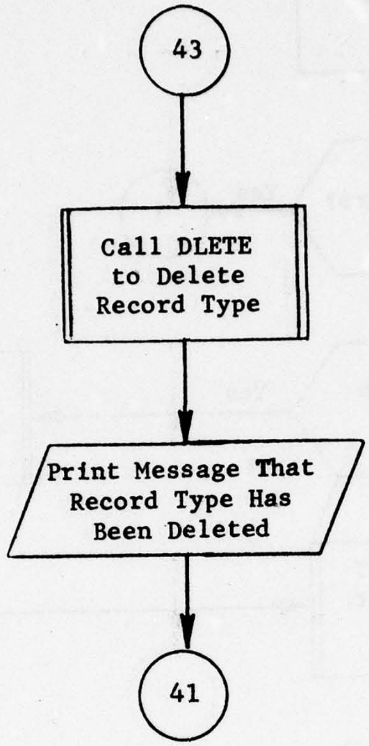



Figure 68. (Part 12 of 12)

3.9.2 Subroutine SYNTAX

PURPOSE: Analyze syntax of input command sentences

ENTRY POINTS: SYNTAX

FORMAL PARAMETERS: None

COMMON BLOCKS: C10, C15, C30, FIRST, IPGT, OOPS, STRING

SUBROUTINES CALLED: HDFND, HEAD, NEXTTT, RETRV

CALLED BY: ERRFND

Method:

This subroutine checks each input string against the current setting of various switches. If the string is one the types of strings expected, the appropriate switches are reset. The following switches and counters are used by SYNTAX:

```
BETWEN - True when processing BETWEEN relation
BLPRCT - Boolean parenthesis level count
BOLTYP - True when expecting relational phrase, left parenthesis
           or NOT (boolean clause only)
BOOL   - True when clause is boolean
CONTEQ - True when an equals relation is being continued in a
           boolean clause
ELEMNT - True when phrases are elemental
ELRT   - 1 - if elemental non-boolean clause
         2 - if elemental boolean clause
ENDCLZ - True if clause may end
ENDCOM - True if sentence may end
EQUALS - True when processing EQUALS relation
INCLZ  - True when processing a clause
INCOM  - True when processing a sentence
LIKE   - True when processing a LIKE relation
PHPNT  - Relational phrase branch
         1 - expecting operation unless inside collection
         2 - expecting value or collection
         3 - check for equal and between continuation
PHPRCT - Relational phrase parenthesis count
RELPHR - True when processing a relational phrase
RESTRC - True when phrase type is restricted relational (EQUALS or
           LIKE)
SINGLE - True when clause type is single
SNGCT  - Count of phrases for single type clause
```

5.5.2 Subroutine GENEDIT. This subroutine (or overlay link) is executed only if adverbs other than ONPRINTS appears within the input command. Its function is purely to drive the other subroutines within overlay link as directed by the type of input clause.

5.5.3 Subroutine BUILDTAB. The edit scheme is constructed within this subroutine. An edit scheme is a set of instructions that informs necessary subroutines how (or in what manner) the data base is to be queried. The philosophy behind this scheme is similar to that outlined for retrieval search schemes (see section 4.4).

5.5.4 Subroutine NORMAL. This subroutine is the third overlay link within EDIT and its function is simply to load default edit schemes.

5.5.5 Subroutine PROCEDIT. Edit schemes, either default or nonstandard, are executed by this subroutine (or overlay).

## 5.6 Edit Internal Common Blocks

The internal common blocks defined for this module are outlined in table 17.

Table 17. EDIT Internal Common Blocks
(Part 1 of 2)

| BLOCK | VARIABLE OR ARRAY | DESCRIPTION |
|---|---|---|
| ATLST | | Used to communicate with ATFNDR utility |
| | LIST(100) | List of attribute identifying numbers |
| | LOCAT(100) | Address of attribute in common C30 |
| | FORMT(100) | Format of attribute<br>1 - Integer<br>2 - Alphabetic<br>3 - Floating Point |
| | LOVAL(100) | Lower limit of attribute in directory |
| | HIVAL(100) | Upper limit of attribute in directory |
| | NUMAT | Number of attributes in LIST, etc. |
| BLOCK | | |
| | BFLDS | Index of start of FIELDS clause in Instruction Code Array |
| | BPRIN | Index of start of ONPRINTS clause in Instruction Code Array |
| | BWHER | Index of start of WHERE clause in Instruction Code Array |
| | BWITH | Index of start of WITH clause in Instruction Code Array |
| CONEDIT | | |
| | NSCH | Number of edit schemes |
| | NFLDS(10) | Number of phrases (fields) in FIELDS clause |
| | NWITH(10) | Number of phrases in WITH clause |
| | IWHERE(10) | Index number of WHERE clause start point |
| IWITH | | |
| | NWATS | Number of phrases in WITH clause |
| | IWITH(50) | Index numbers of start points of phrases in WITH clause |
| LOCATER | | |
| | LBLOCK(47) | Contains information encoded by FORMLOC. Used to print error messages (format, addresses of output variables and number of variables) |
| NAMES | | |
| | NAMAT(100) | Attribute names |
| | NLIST(100) | List of legal values for 'LIST' attributes |
| PRINSP | PRINON | Optional Print switch<br>True = produce optional print<br>False = do not produce optional print |

Table 17. (Part 2 of 2)

| BLOCK | VARIABLE OR ARRAY | DESCRIPTION |
|---|---|---|
| RTLST | | Used to communicate with ATFNDF utility |
| | RTLIST(100) | List of record type numbers |
| | NYMREC | Number of record types in RTLIST |
| | HDREC | Record type name of primary header |
| | HCLASS | CLASS value assigned for primary header |
| | HSIDE | SIDE value assigned for primary header |
| | HOPT | Retrieval scheme header option |
| | JHDR | Record type number of primary header |
| SCHEME | | |
| | POINT | Retrieval scheme pointer to next instruction in scheme |
| | SCHEME(200) | Retrieval scheme |
| SIDES | SIDES(5) | List of legal values for SIDE (derived by FINDSIDE) |

### 5.7 Subroutine ENTMOD

**PURPOSE:** Entry subroutine for EDIT module

**ENTRY POINTS:** ENTMOD (first subroutine called when overlay EDIT is executed

**FORMAL PARAMETERS:** None

**COMMON BLOCKS:** None

**SUBROUTINES CALLED:** COUNTS, GENEDIT, INSGET, NORMAL PROCEDIT

**CALLED BY:** MODGET

**Method:**

First the verb is validated. Next, the input is checked for adverbs. If no adverb other than ONPRINTS was input, the normal-or-default edit scheme is used (NORMAL is called). Otherwise the edit scheme is to be specified by the input and GENEDIT is called. In either case, PROCEDIT is called to perform the edits according to the scheme and, finally COUNTS is called to check on the data base limits.

Subroutine ENTMOD (EDIT) is illustrated in figure 69.

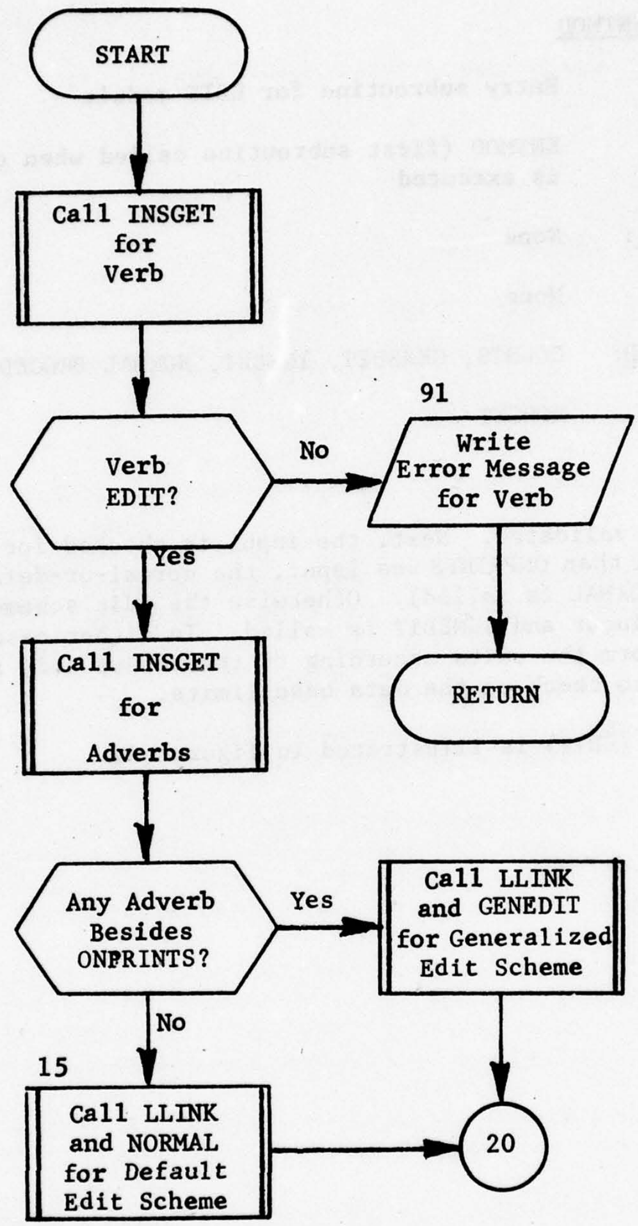



Figure 69. Subroutine ENTMOD (EDIT) (Part 1 of 2)

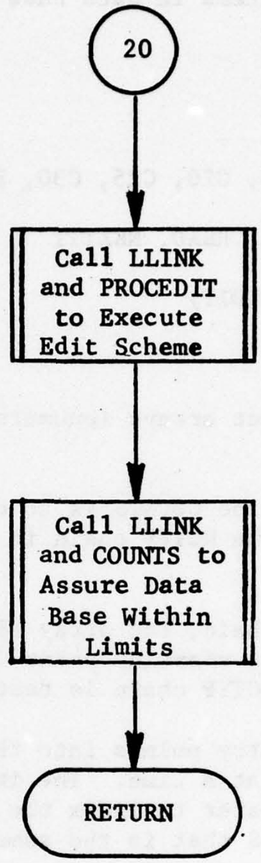



Figure 69. (Part 2 of 2)

5.8 Subroutine COUNTS*

PURPOSE: Counts items in data base to see if within limits

ENTRY POINTS: COUNTS

FORMAL PARAMETERS: None

COMMON BLOCKS: C10, C15, C20, C25, C30, SIDES

SUBROUTINES CALLED: FINDSIDE, HEAD, NEXTTT

CALLED BY: ENTMOD (EDIT)

Method:

COUNTS uses a number of data set arrays (enumerated later) to generate a census of the data base.

The first execution of subroutine COUNTS is to call FINDSIDE to retrieve the SIDEs in the data base. The RCTYP chain in the Data Organization Index is also headed.

For each record on the RCTYP chain, the array HEADER is scanned to see if this record can be used as a starting point into the Data Base. If not, the next record on this RCTYP chain is tested.

If the record is one of the entry points into the data base the MYNAMZ chain is processed one record at a time. The index into the HEADER array is saved as it is used later to index the other control arrays. The index to the value in SIDES that is the same as XSIDE is also saved.

The header, whose reference code was found in the Data Organization Index, is retrieved. The header index is now used to find which chain in CHAIN is to be traversed. The value of RNUMB, from the Data Organization Index, will determine when the chain is completed.

For each record on this chain, the count is incremented and the chain in array CHAIN2, indicated by the header index, is processed.

The corresponding value in array EOCH will determine when the chain is completed.

When the processing of this chain is completed the values are compared to see if the counts are less than the value in array MAX2, indicated by the header index. Processing then continues with the processing of the next record on CHAIN.

When the end of the CHAIN chain is encountered, the counts are checked against the corresponding value in array MAX.

* Main routine of overlay link ECOUNT.

The next record on the MYNAMZ chain is processed. When the end of the MYNAMZ chain is reached the next record on the RCTYP chain is examined. This continues until the RCTYP chain is complete.

There are a number of points in this processing in which special processing occurs. These points are flagged by the doubly dimensional array SPECIAL. The first index is the level and the second is the header index.

Level one occurs after the header has been retrieved. The only special processing here is to count the classes for each side.

Level two is when the chain immediately under the header is being processed. There are three cases at this point that require special processing.

First, only unique target types are to be counted. Second, in order to check the number of weapons in a group it is necessary to traverse the WPINGP and MYPAY chains to find the payload size and then the WEPPAY and WARHED chains to find out if it is a weapon. Finally, a count of ASMs is made concurrently with the warhead counts.

The third level is in processing the second chain (CHAIN2). There are six special cases listed here: 1. the vehicle count in a group must be a summation not a simple count, 2. only unique penetration and depenetration legs are to be counted, additionally the Recovery Bases for Depenetration corridors are checked at this time, 3. only complexes are examined, not the simple or multiple targets, 4. only bombs and SAMs are to be counted in the payload, 5. only unique weapon types are counted, and 6. tanker bases are counted (requires traversing one extra chain).

Figure 70 illustrates COUNTS.

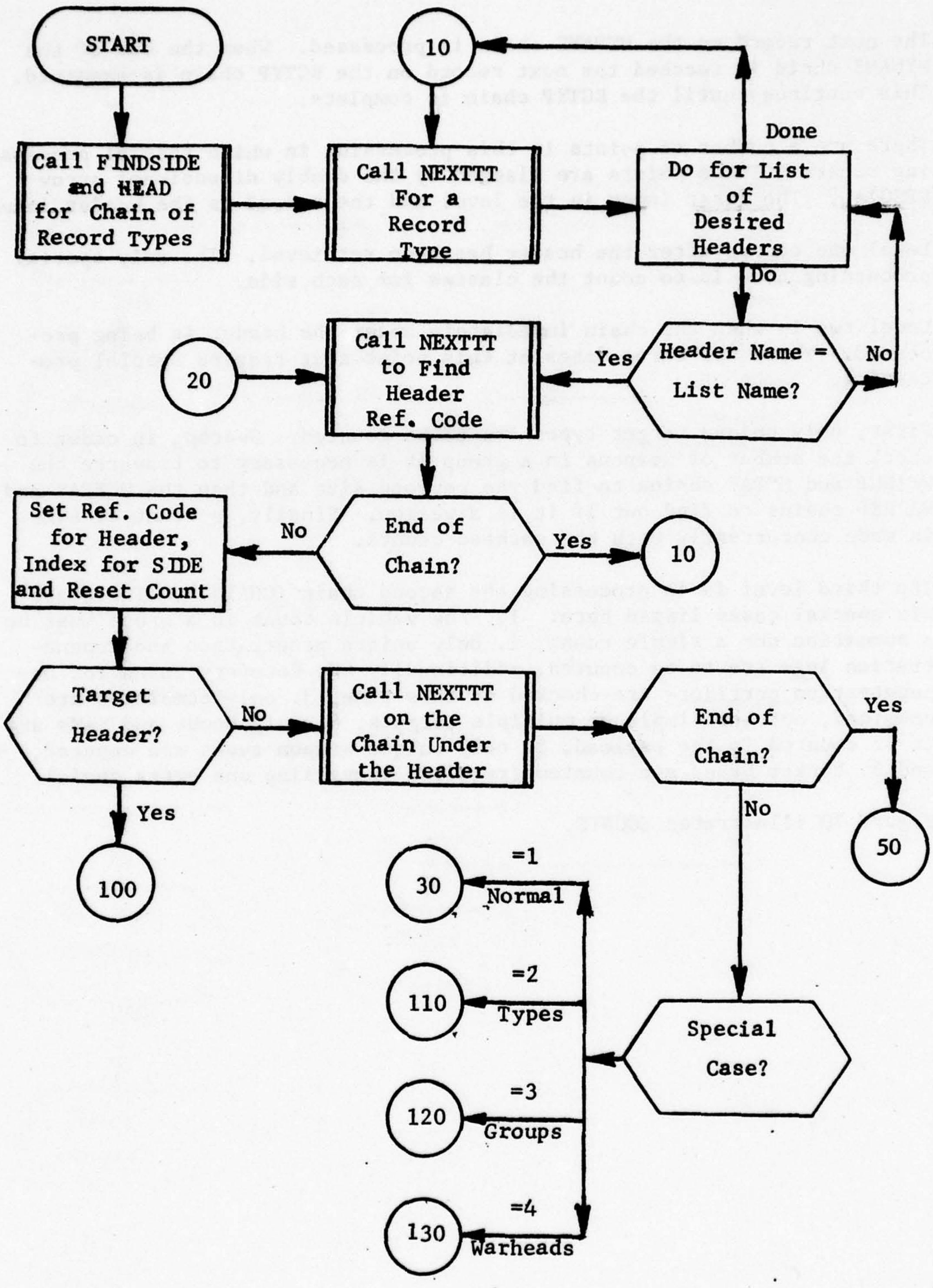



Figure 70 Subroutine COUNTS (Part 1 of 7)

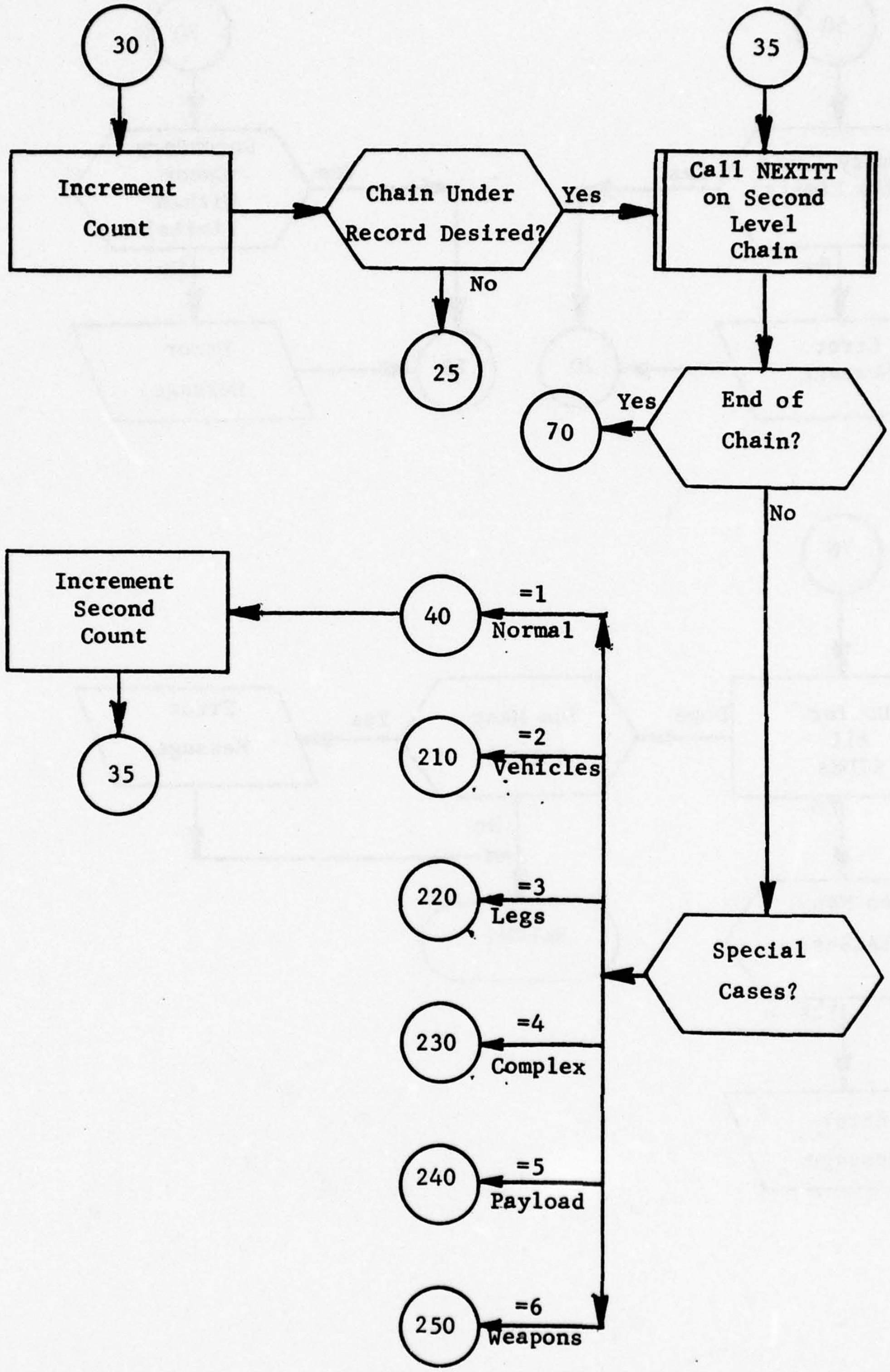



Figure 70. (Part 2 of 7)

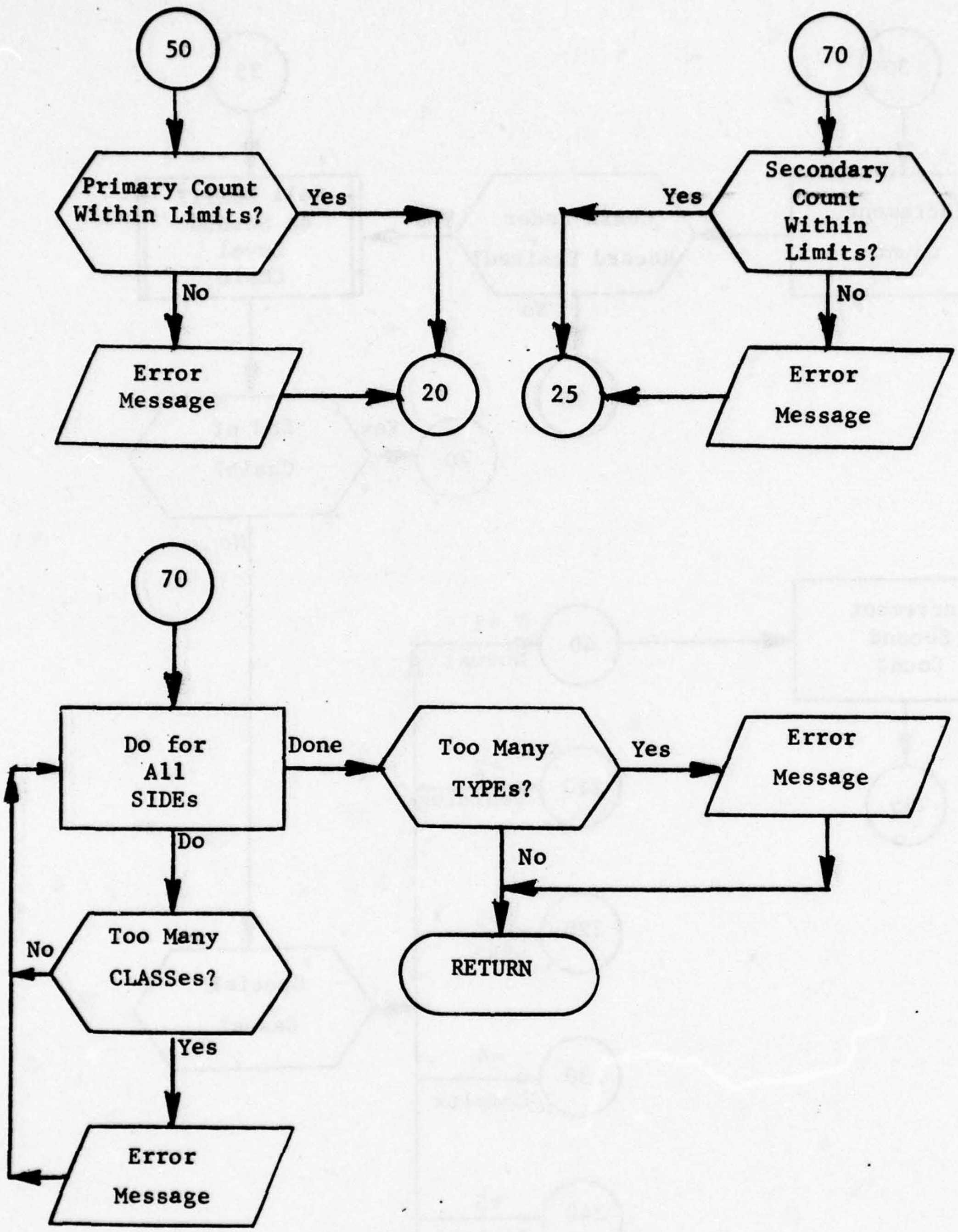



Figure 70 (Part 3 of 7)

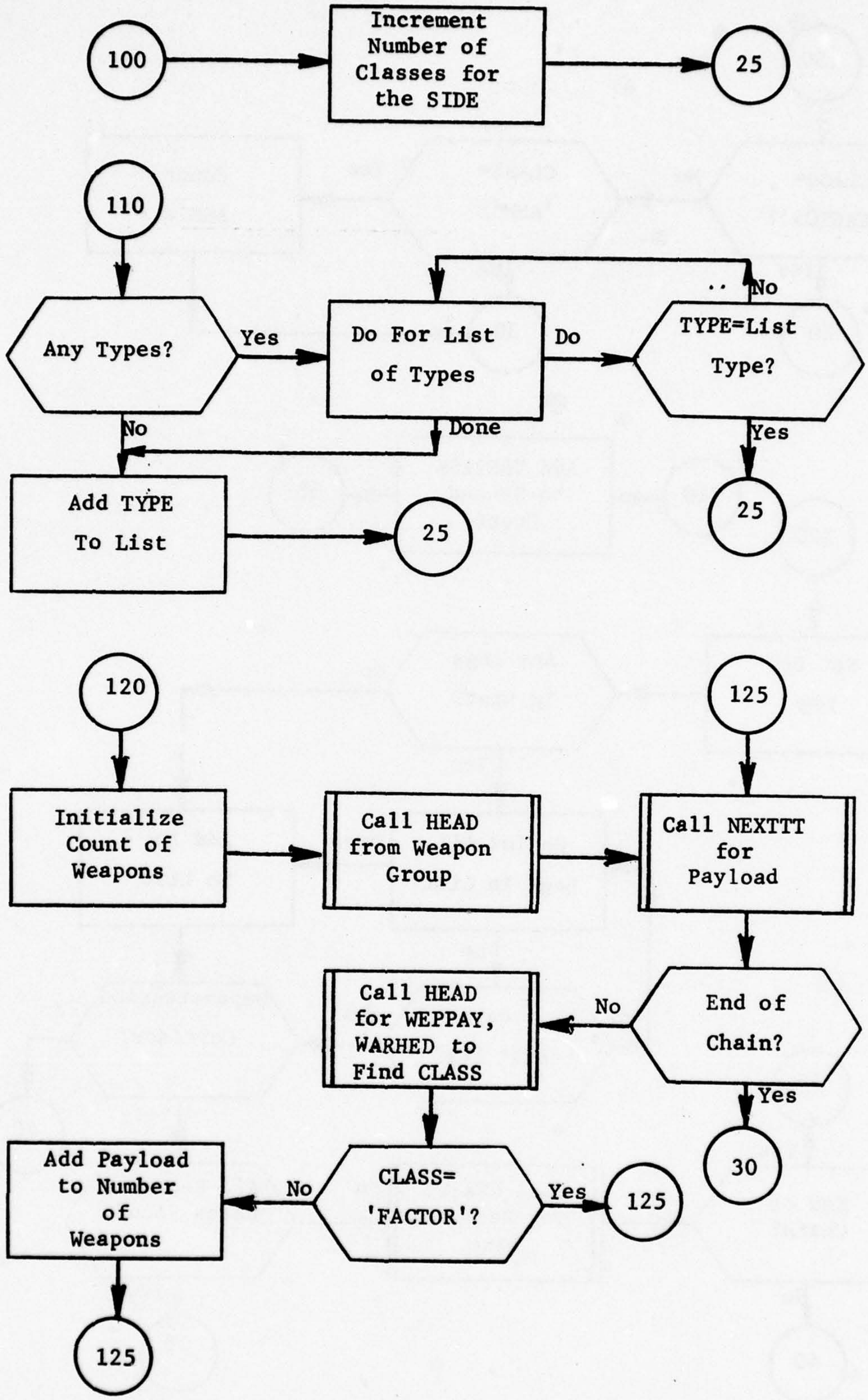



Figure 70 (Part 4 of 7)

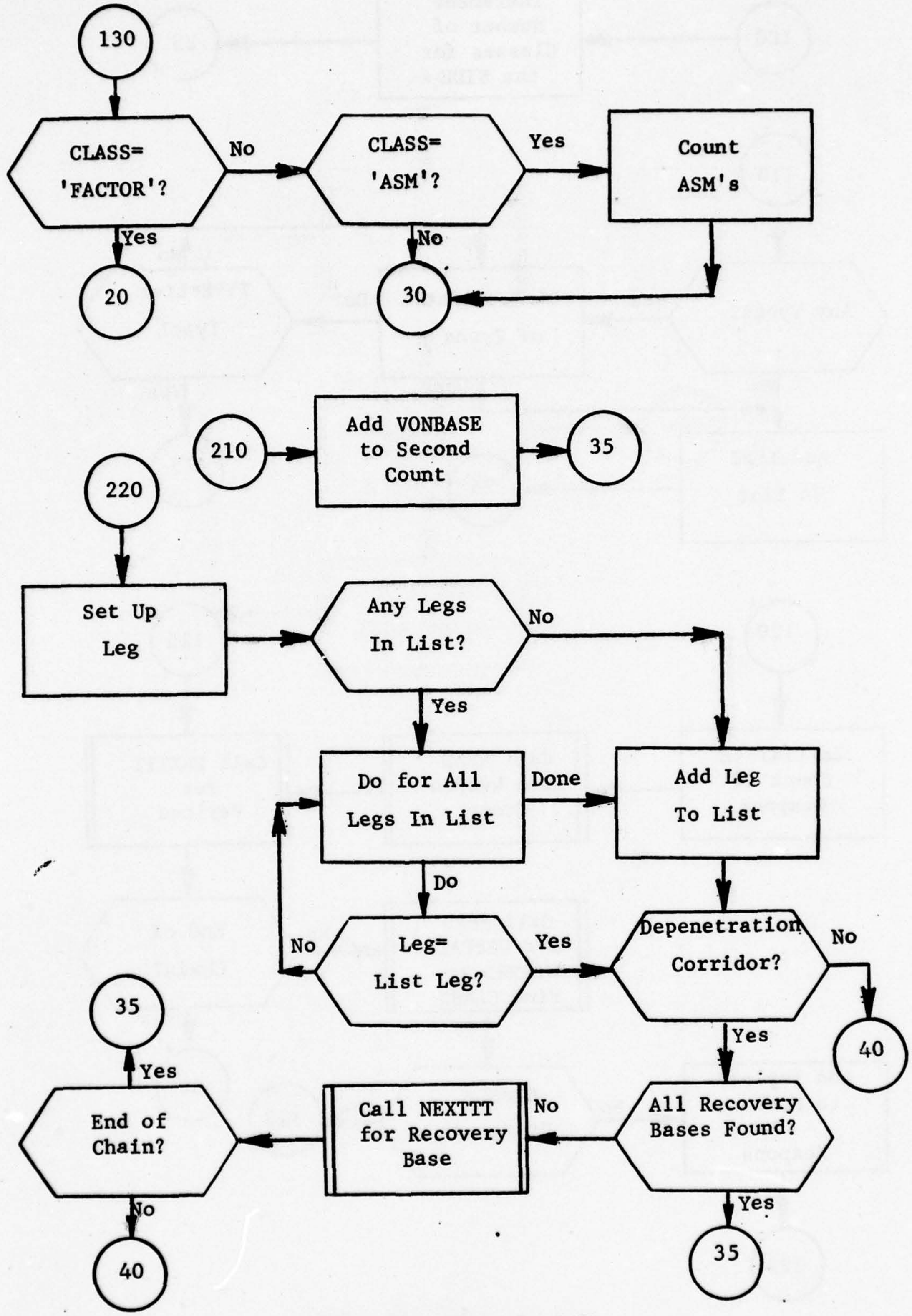



Figure 70 (Part 5 of 7)

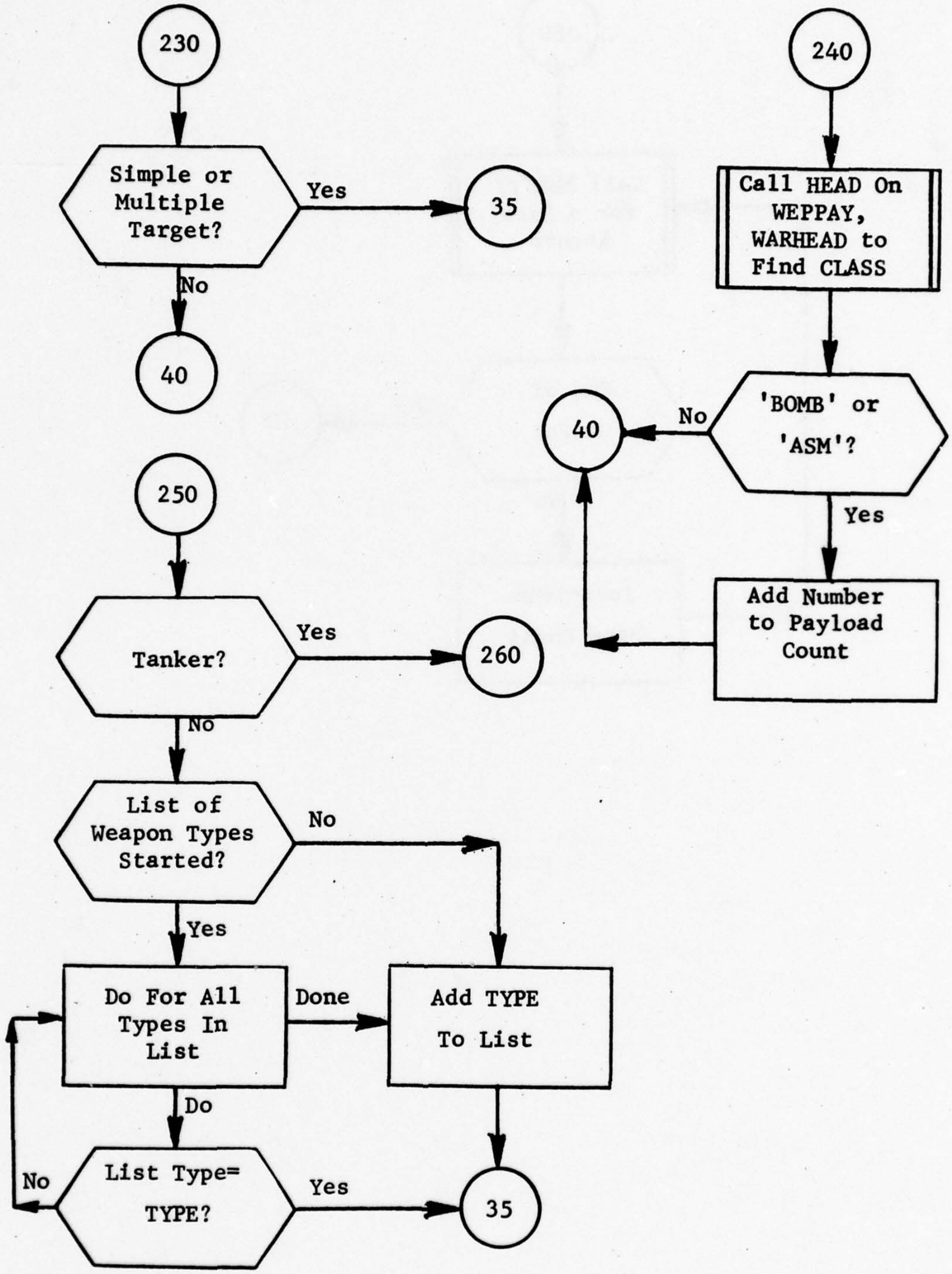



Figure 70 (Part 6 of 7)

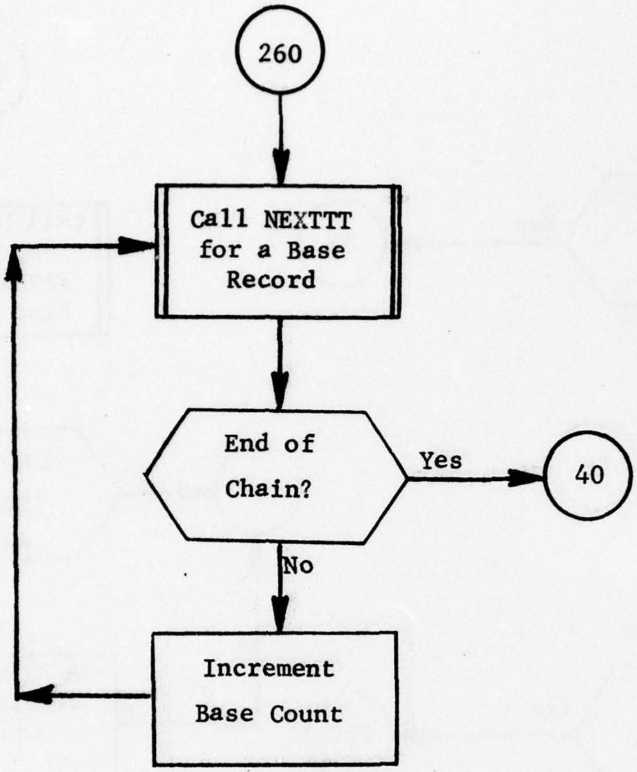



Figure 70 (Part 7 of 7)

## 5.9 Subroutine GENEDIT*

PURPOSE: Edit scheme construction drives

ENTRY POINTS: GENEDIT

FORMAL PARAMETERS: None

COMMON BLOCKS: BLOCK

SUBROUTINES CALLED: BUILDTAB, INSGET

CALLED BY: ENTMOD (EDIT)

Method:

First INSGET is called for the number of adverbs. Then the adverbs are checked (and validated), one at a time until the end of the set of adverbs or until a duplicate adverb is encountered. As each adverb is read, its clause pointer (POINTER) is saved in common block BLOCK.

If a duplicate adverb is encountered BUILDTAB is called for the current set of adverbs and then the pointers (START) are reset. At the end of the full set of adverbs BUILDTAB is called for the last subset.

Subroutine GENEDIT is illustrated in figure 71.

---

* Main routine of overlay link EGENED.

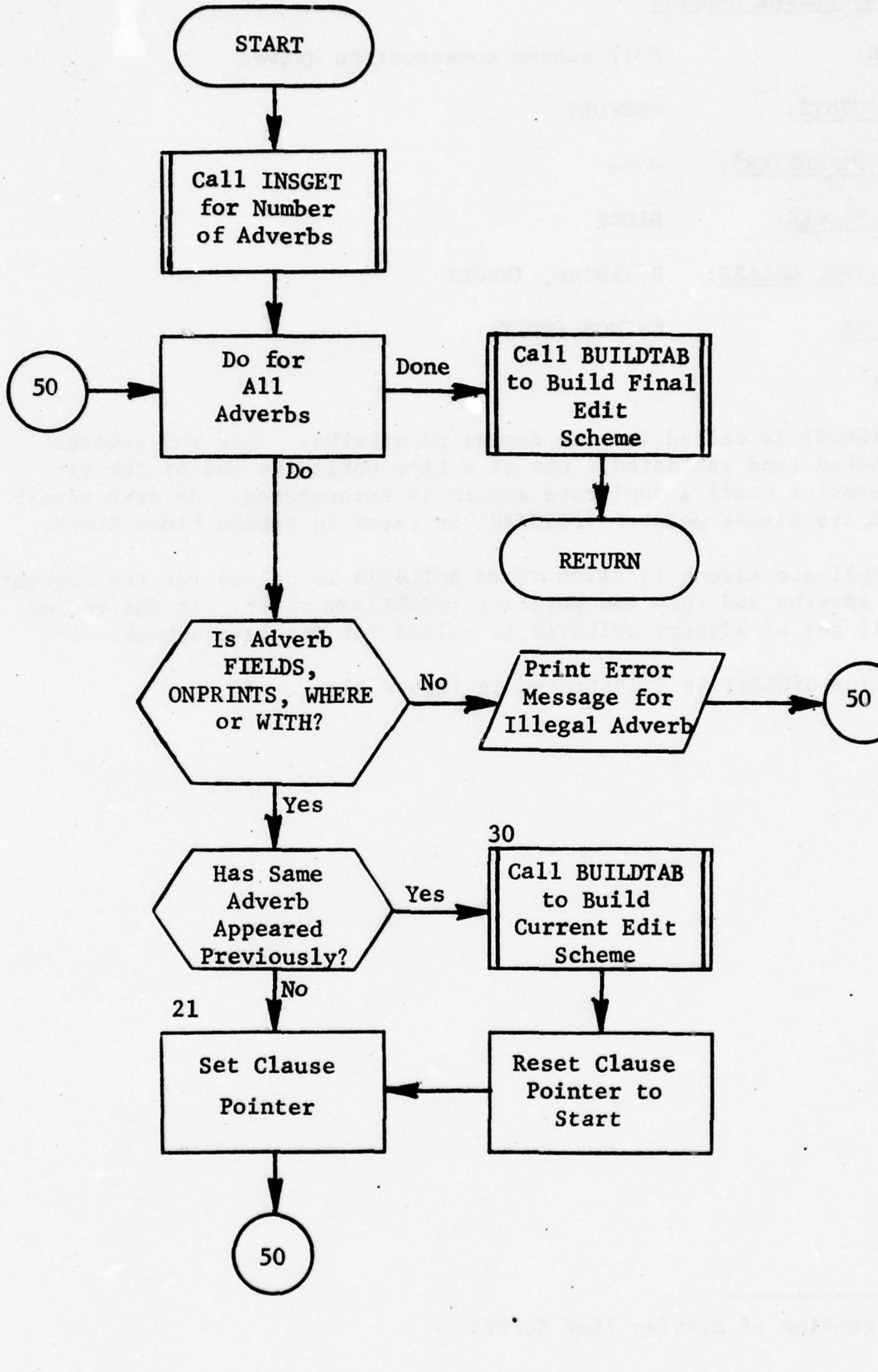



Figure 71. Subroutine GENEDIT

## 5.9.1 Subroutine BUILDTAB

PURPOSE: To construct an edit scheme

ENTRY POINTS: BUILDTAB

FORMAL PARAMETERS: None

COMMON BLOCKS: ATLST, BLOCK CONEDIT, IWITH, LOCATER, NAMES, PRWSP, RTLIST, SCHEME

SUBROUTINES CALLED: ATFNDR, FORMLOC, INSGET, LINKUP SETFLD, SETSCH, SWHERE, SWITH

CALLED BY: GENEDIT

Method:

The first operation for BUILDTAB is to reset any pointers or counts that may have been set in an earlier call. Then the individual clauses are processed.

If there is a FIELDS clause,it is processed in the following manner. First subroutine SETFLD is called to build the arrays of the value ranges, attribute names, and format. These values are passed in commons ATLIST and NAMES. These arrays are written on a scratch file (unit 21) for later processing by PROCEDIT and if the optional prints are desired, they are printed.

If there is a WITH clause, it is first processed by subroutine SWITH which will collect attributes to build the Search Scheme (see description of Retrieval Scheme in section 4.4) and set pointers to the beginning of the phrases within the clause. These pointers and the WITH clause are then saved on the scratch file (unit 21) in 50 word blocks, and optionally printed.

If there is a WHERE clause it will be scanned for attributes to help build the Search Scheme through a call to SWHERE.

Subroutine ATFNDR, LINKUP, and SETSCH are called in order to construct a scheme to examine the data base. ATFNDR finds the records containing the attributes found previously by SETFLD, SWITH and SWHERE. ATFNDR adds any additional records to make a continuous chain. SETSCH then uses this list to set up a sequence of operations to be executed by GETNXT, in PROCEDIT. These three routines are described in detail in the Utilities section.

Since, simply knowing that an error exists somewhere in the data base is in sufficient, FORMLOC is called to create a message that will describe where in the data base the error was found. There is also an optional print available for this data.

The scheme for searching the data base and the locator information are then written on the scratch file.

Figure 72 illustrates BUILTAB.

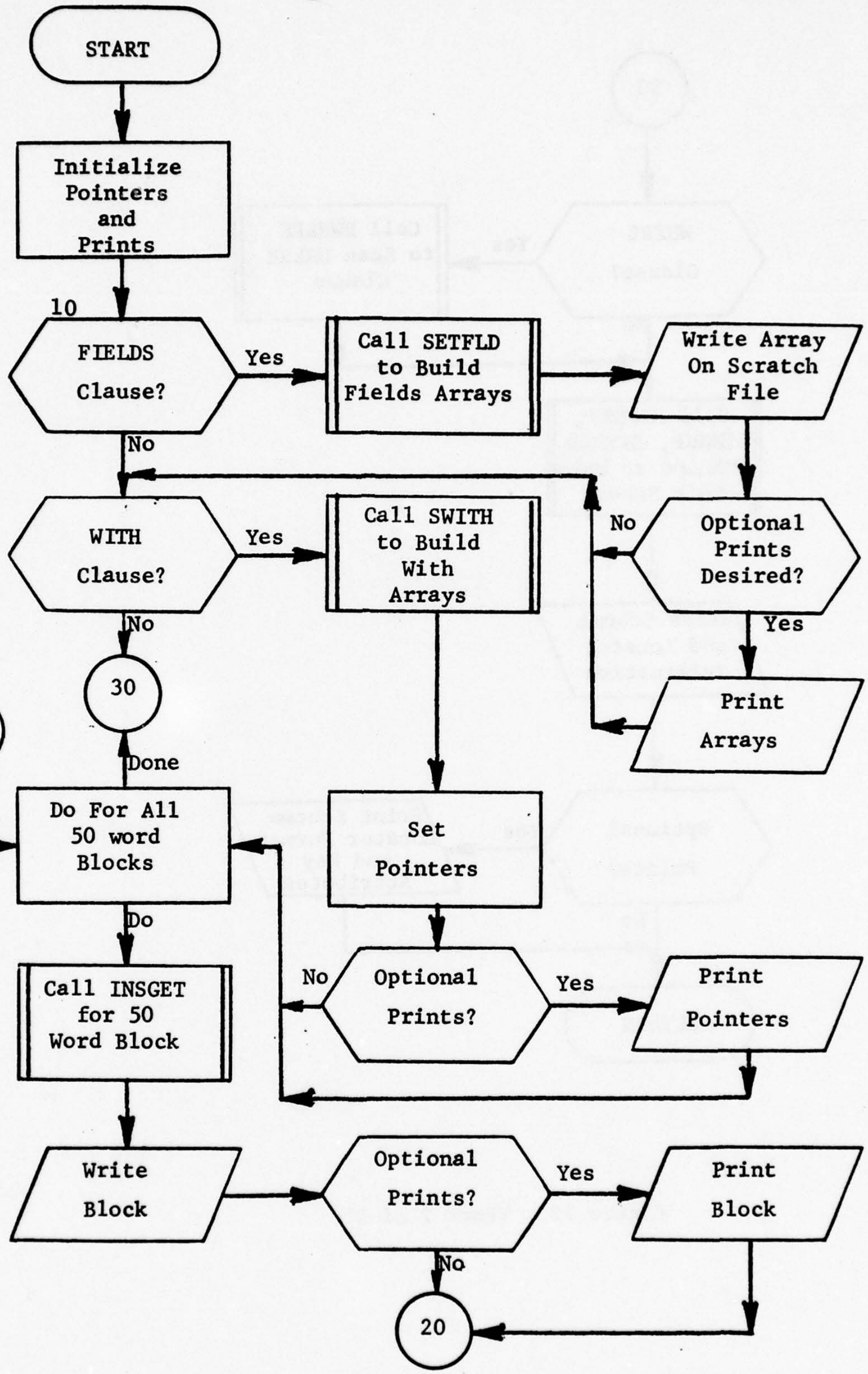



Figure 72. Subroutine BUILDTAB (Part 1 of 2)

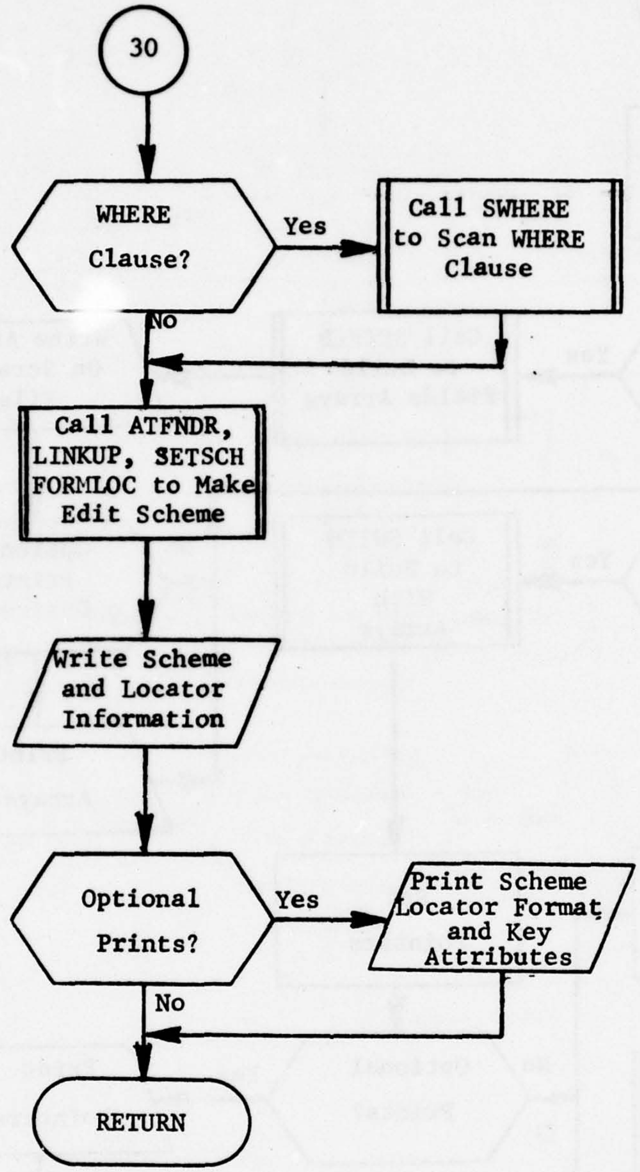



Figure 72. (Part 2 of 2)

5.9.2 Subroutine FORMLOC

PURPOSE: Produces a record to find where an error occurs

ENTRY POINTS: FORMLOC

FORMAL PARAMETERS: IREC: An array of record numbers
MREC: The number of items in IREC

COMMON BLOCKS: LOCATER

SUBROUTINES CALLED: CONCAT

CALLED BY: BUILDTAB

Method:

There are five arrays that are used in constructing the message used to locate an error. The arrays are:

- o FORMS: contains all 11 segments that can appear in the locater message.

- o FSTART: contains the beginning character in FORMS for each attribute. FSTART(12) is just beyond the eleventh segment.

- o LENOUT: contains the output length of each segment. This is used to determine is more than one line will be needed to hold the message.

- o KEYREC: contains up to three record numbers associated with this attribute.

- o KEYLOC: contains the location in COMMON C30 of the attribute.

FORMLOC uses these values to generate the format statement in LOCATER, the C30 locations in KEYATT and the number of key attributes in NKEYAT.

The processing is illustrated in figure 73.

Note that the CLASS and SIDE are always used. For each of the key attributes the array IREC is scanned to see if any entry matches one of the three values in KEYREC. If so, the position of the data base pointed to by FSTART is moved to LOCATER using the system routine CONCAT. If this causes a multiline entry, then the format in LOCATER is adjusted to account for the two lines of output. The location in common C30 of the attribute is then added to KEYATT and NKEYAT is incremented.

This continues for all key attributes.

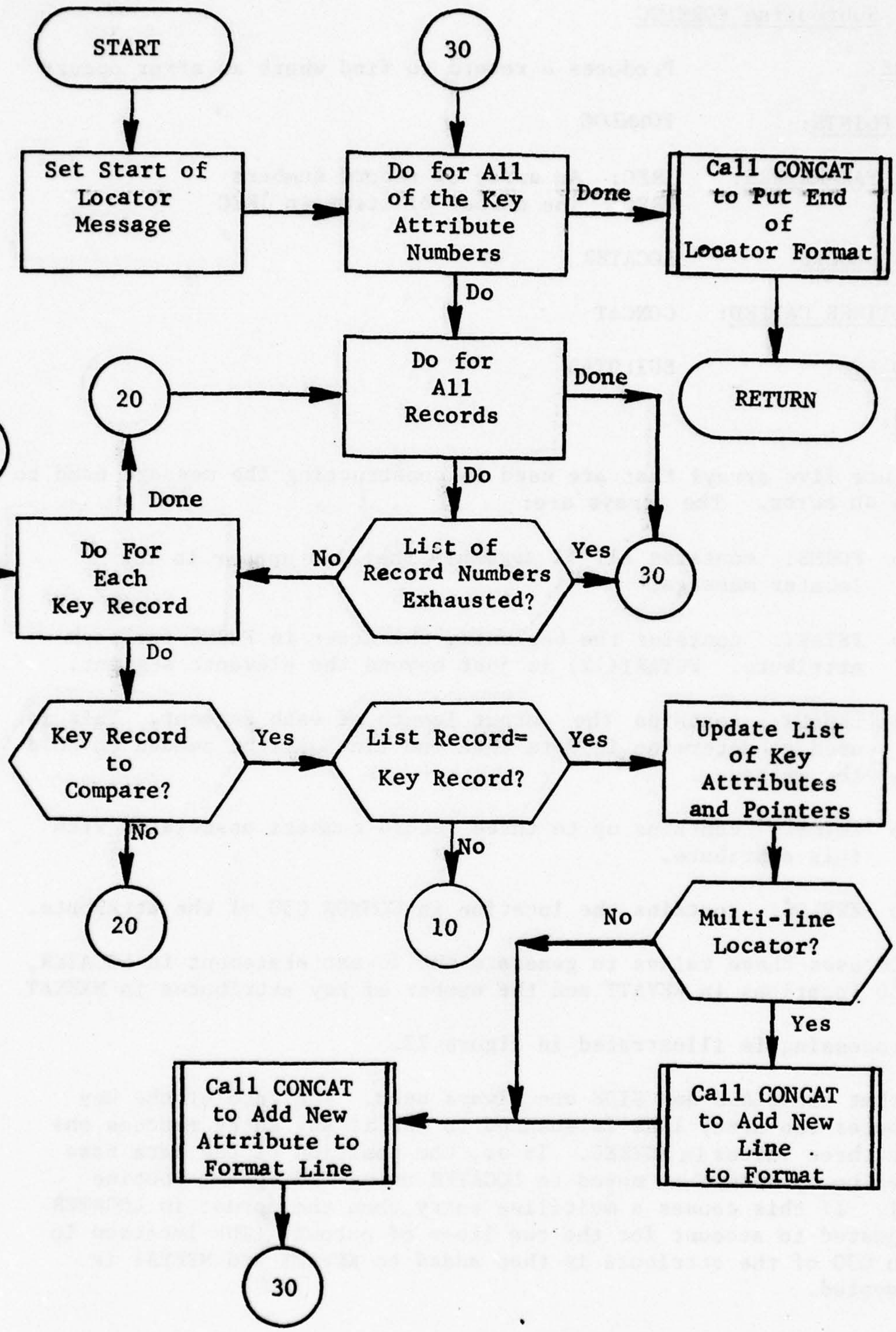



Figure 73. Subroutine FORMLOC

5.9.3 Subroutine SETFLD

| | |
|---|---|
| PURPOSE: | To scan the FIELDs clause collecting the attributes, their location, name, and legal values |
| ENTRY POINTS: | SETFLD |
| FORMAL PARAMETERS: | START: Beginning of the FIELDs clause<br>N: The number of phrases in the FIELDs clause (output) |
| COMMON BLOCKS: | ATLST, C10, C30, NAMES |
| SUBROUTINES CALLED: | ALOG, HEAD, INSGET, NEXTTT |
| CALLED BY: | BUILDTAB |

Method:

The processing in subroutine SETFLD takes place in two steps. The first is the collection of attribute numbers from the input. The second is the assigning of necessary values for each of these attributes.

The first section consists simply of calling INSGET for five word segments containing the attribute number. If it is not already in ATNUMB it is added to the attribute list.

The second section consists of examining the ATRIB chain in the Data Organization Index until all attributes in the list built in section one have been defined.

For each record on the ATRIB chain the attribute is checked against ATNUMB. If the attribute is found, the value range (LOVAL and HIVAL), the format (FORMT) and the name (NAMAT) are saved. Next, a check is made to see if the attribute has a list of legal values. If so, HIVAL is redefined as the starting position in the array of legal values (LIST - common block NAMES). LOVAL will be the last position in this array. All legal values are added to LIST, adjusting LOVAL for each new entry.

Processing them continues for the rest of the ATRIB chain.

Figure 74 describes SETFLD.

Note that the common block ATLST is redefined at this point. ATNUMB is the same but the remainder of the common block has been redefined to hold the attributes location in common C30 (LOCAT), the format for printing the attribute (FORMT), the low value or high index into the list (LOVAL), and the high value or low index into the list of legal values (HIVAL). Common NAMES holds the remaining information, the twelve character attribute name (NAMAT) and the list of legal values (LIST).

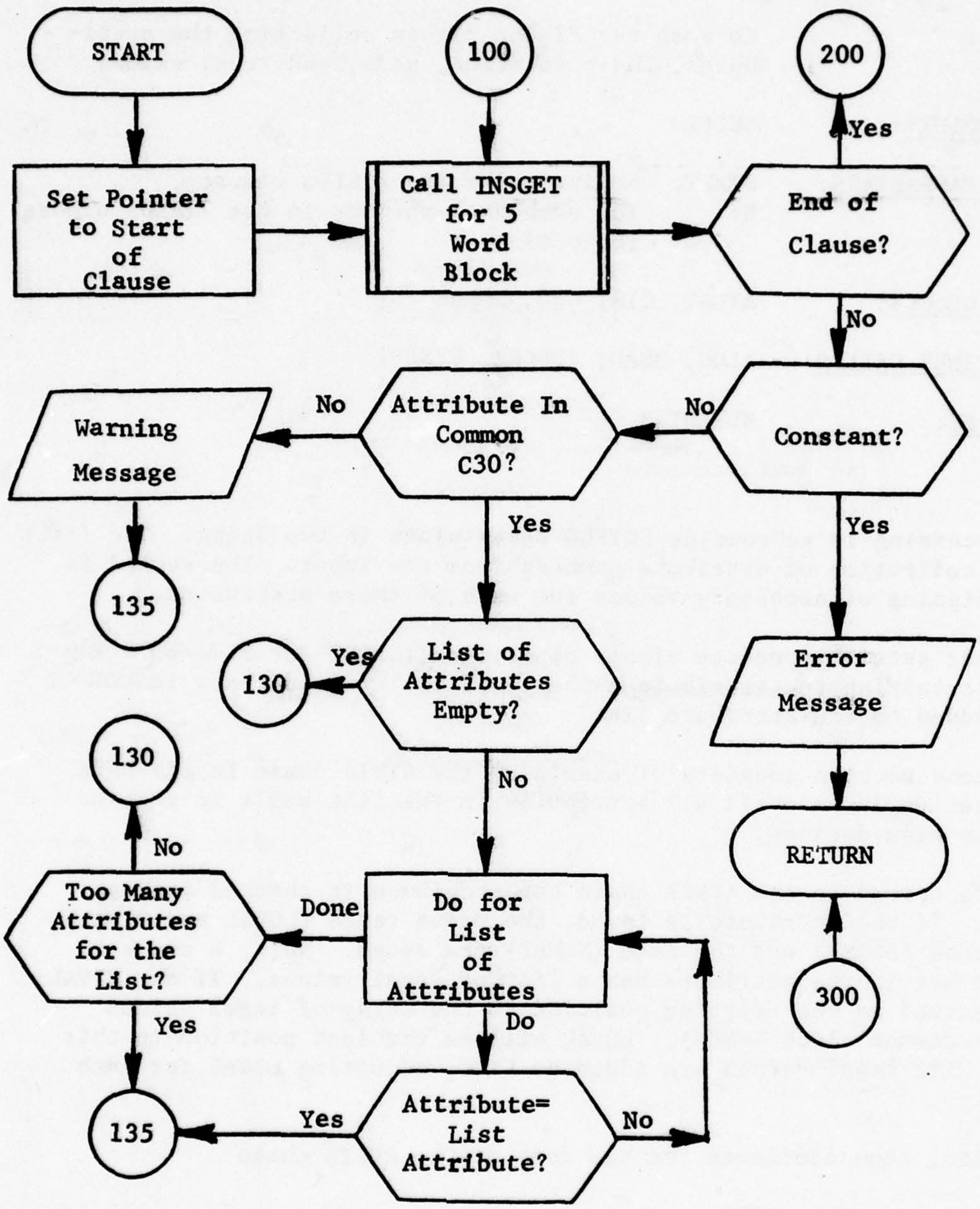



Figure 74. Subroutine SETFLD (Part 1 of 4)

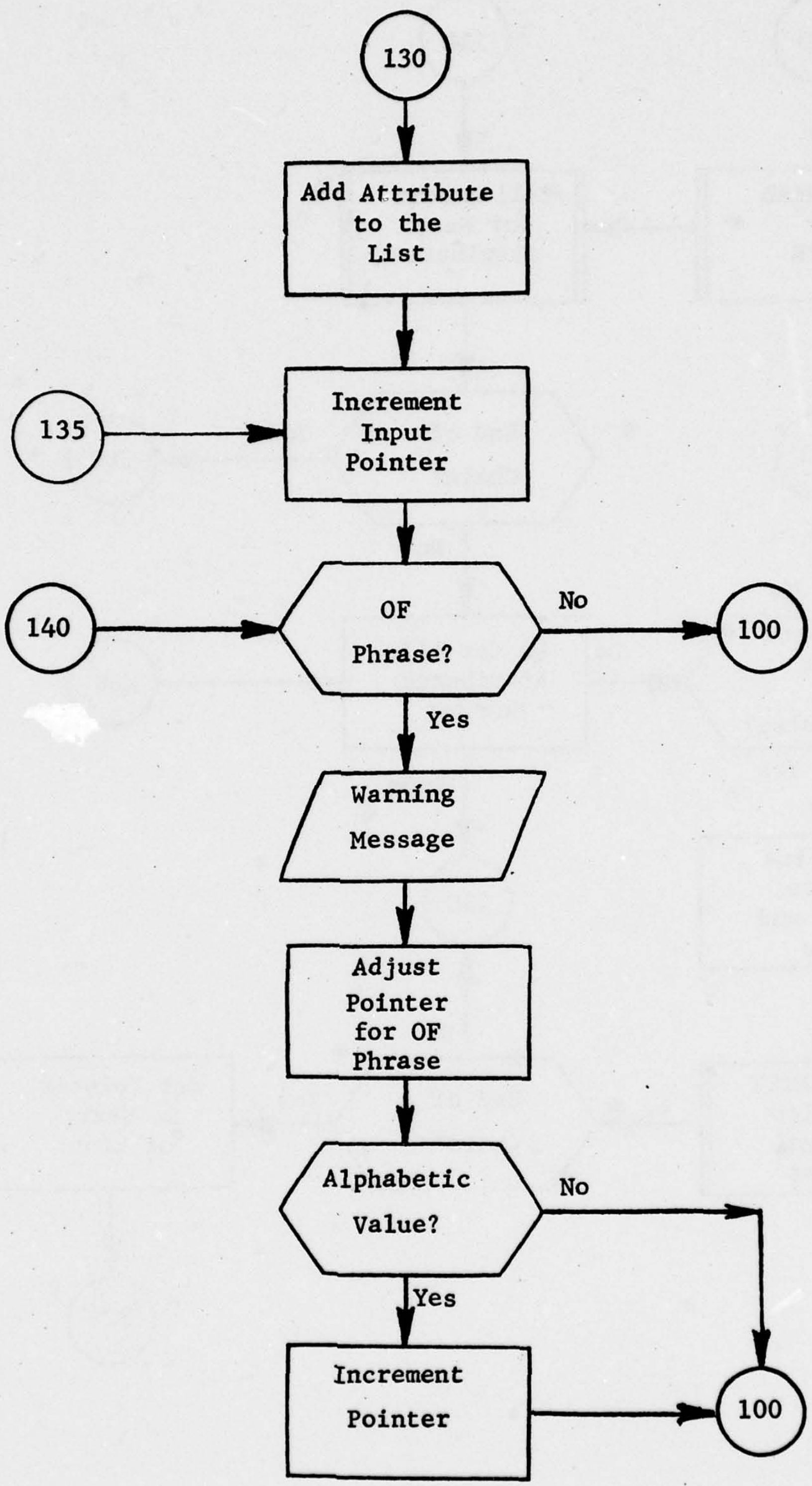



Figure 74. (Part 2 of 4)

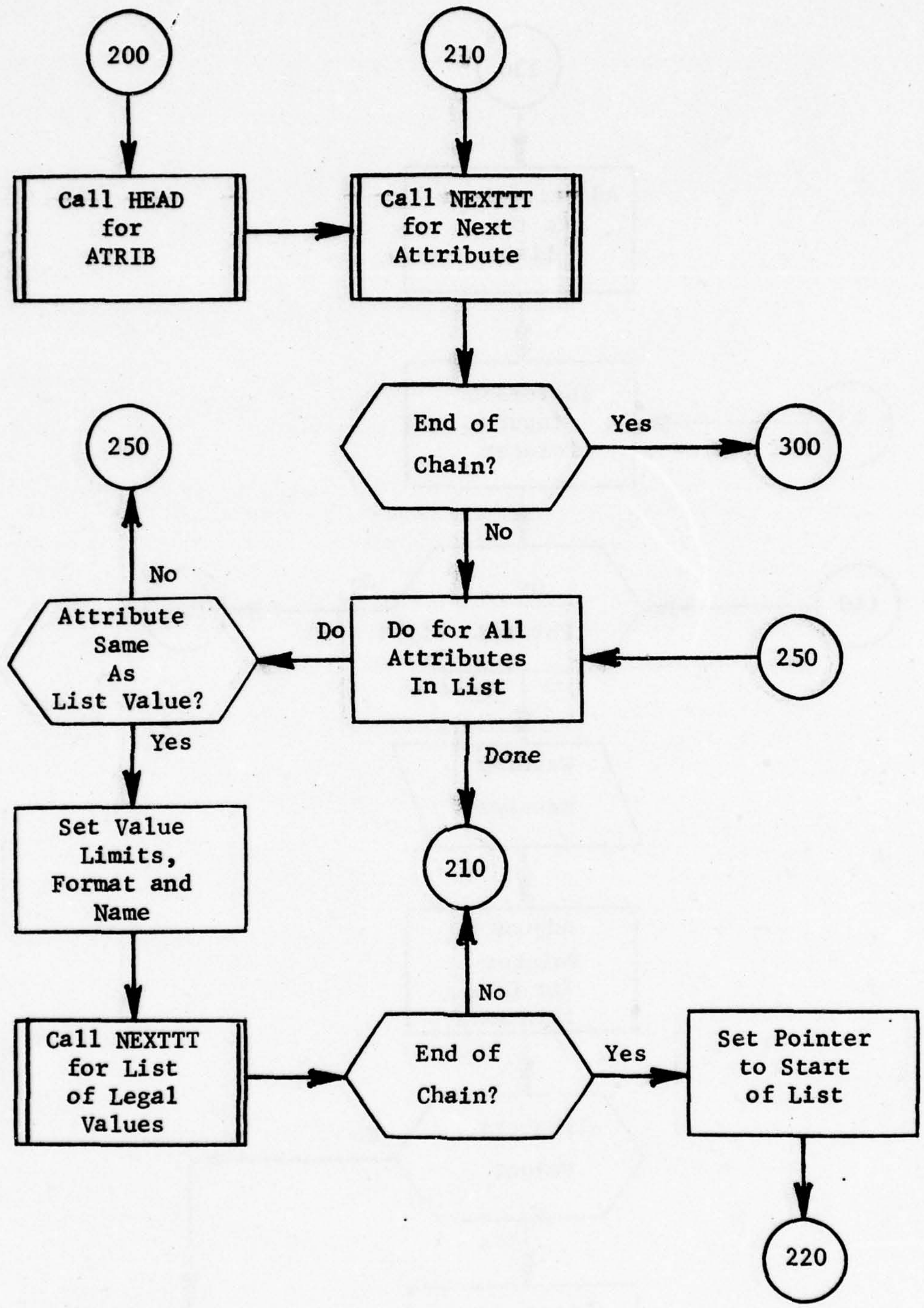



Figure 74. (Part 3 of 4)

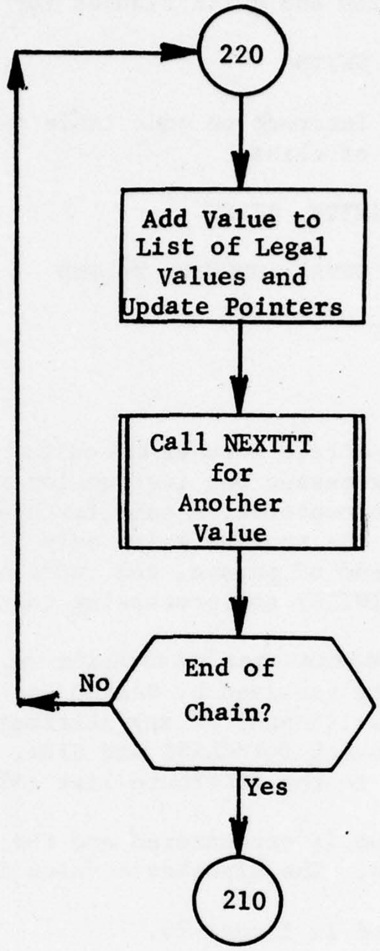



Figure 74. (Part 4 of 4)

5.9.4 Subroutine SWITH

PURPOSE: Scans WITH and WHERE clauses for attributes

ENTRY POINTS: SWHERE, SWITH

FORMAL PARAMETERS: BEGIN: Instruction code table index for beginning of clause

COMMON BLOCKS: ATLST, IWITH, RTLST

SUBROUTINES CALLED: INSGET, OFVAL, UNCODE, VALFND

CALLED BY: BUILDTAB

Method:

First switch WITH is set to indicate whether the call was for WITH or WHERE clause. Then the clause is processed one instruction at a time. When a terminator instruction is encountered, a call for a WHERE clause ends. For a call for a WITH clause, the routine exits only if the terminator is an end of clause. For an end of phrase, the index of the beginning of the next phrase is saved (IWITH) and processing continues.

For general instructions the code branches based on the type of instruction. OF phrases and LIKE strings are resolved by VALFND and stored by OFVAL. The number (DANUMB) and address (DAADD) of any attribute is saved for use with LIKE strings and to check for CLASS and SIDE. Furthermore, the attribute number is added to the attribute list (ATNUMB).

Finally, if an alphabetic value is encountered and the previous attribute (DAADD) was CLASS or SIDE. The alphabetic value is saved.

Subroutine SWITH is illustrated in figure 75.

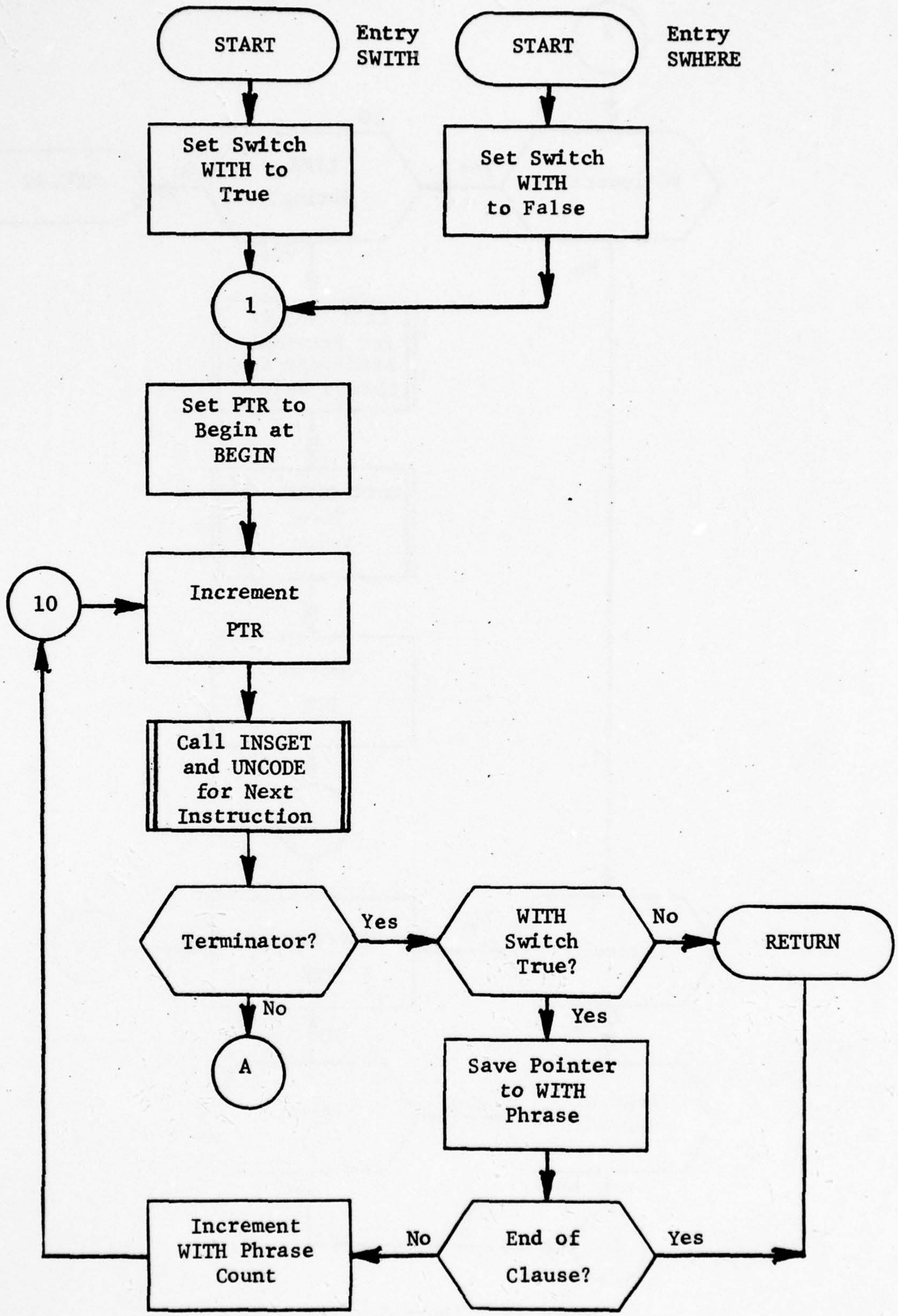



Figure 75. Subroutines SWITH: Entries SWITH and SWHERE
(Part 1 of 4)

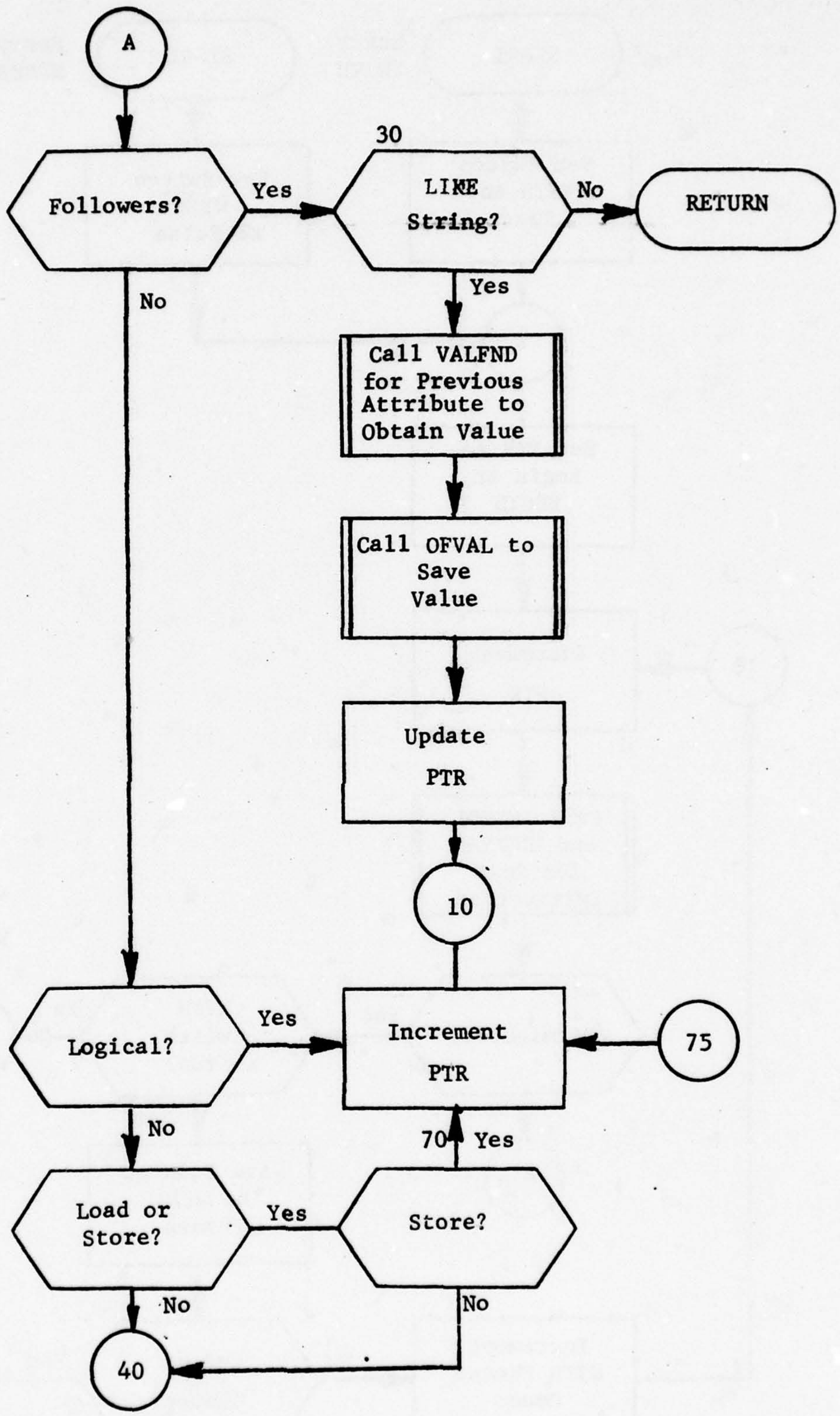



Figure 75. (Part 2 of 4)

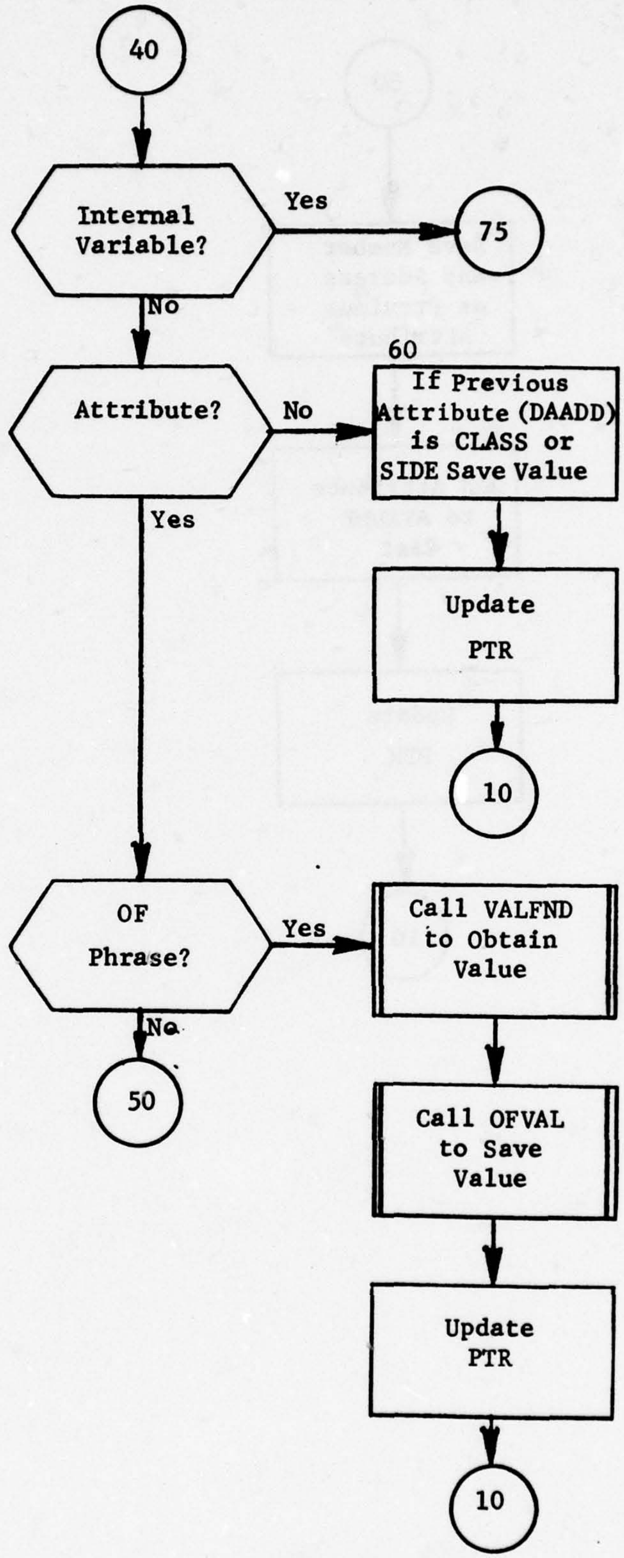



Figure 75. (Part 3 of 4)

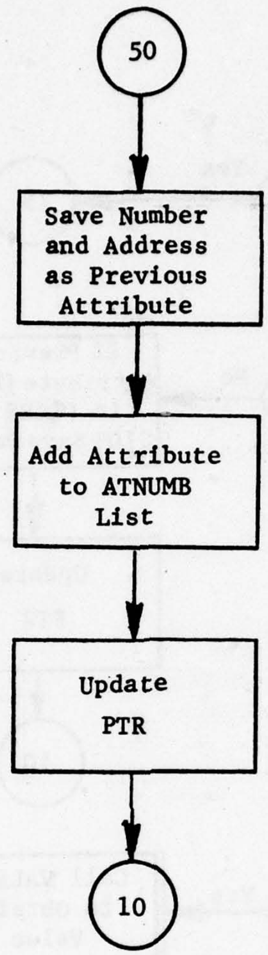



Figure 75. (Part 4 of 4)

## 5.10 Subroutine NORMAL*

PURPOSE: Loads the default edit scheme

ENTRY POINTS: NORMAL

FORMAL PARAMETERS: None

COMMON BLOCKS: CONEDIT

SUBROUTINES CALLED: None

CALLED BY: ENTMOD (EDIT)

Method:

Subroutine NORMAL puts the WITH clause information, the editing scheme, and the locater information onto the scratch file for each of the five default edit passes.

The format for this output is identical to that produced by subroutine BUILDTAB and will be treated by PROCEDIT as any other set of editing schemes.

Figure 76 illustrates Subroutine NORMAL.

* Main routine of overlay link ENORMA.

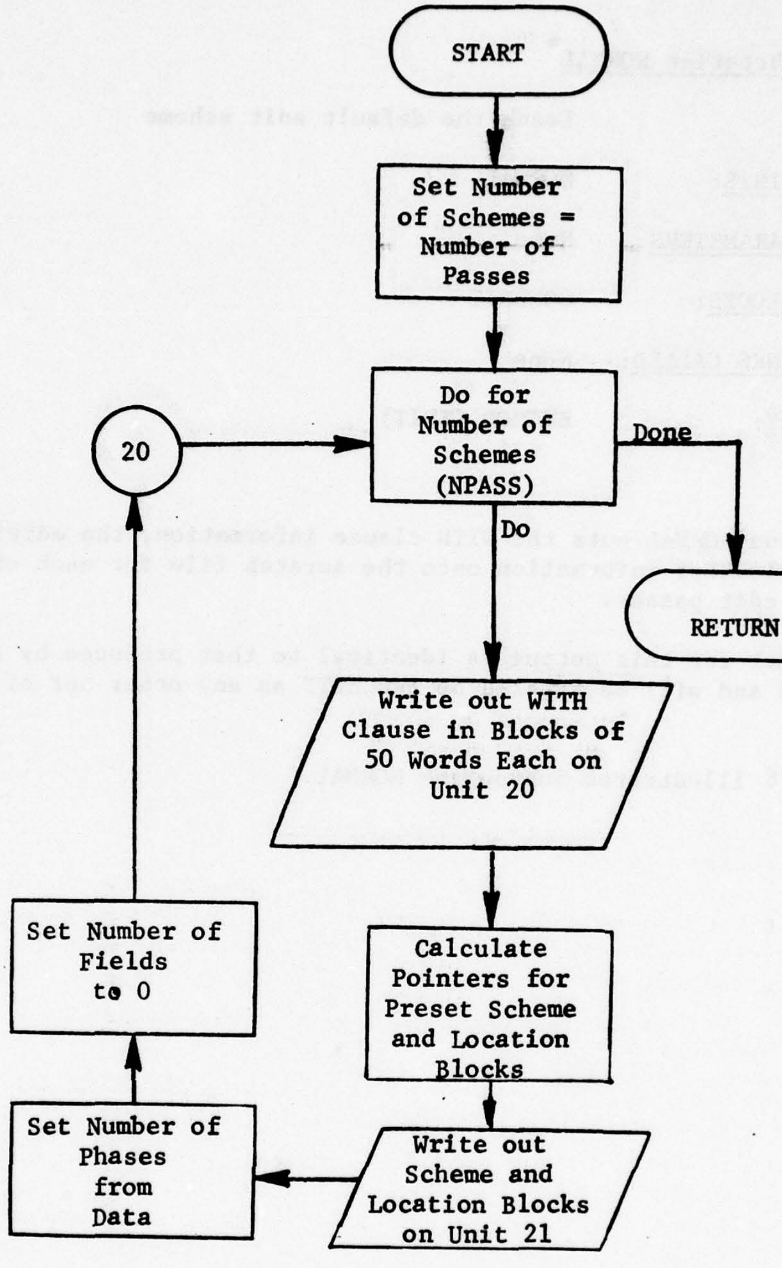



Figure 76. Subroutine NORMAL

## 5.11 Subroutine PROCEDIT*

PURPOSE: To execute the edit scheme

ENTRY POINTS: PROCEDIT

FORMAL PARAMETERS: None

COMMON BLOCKS: C30, CONEDIT, LOCATER

SUBROUTINES CALLED: GETNXT, OFVAL, XWHERE, XWITH

CALLED BY: ENTMOD (EDIT)

Method:

Subroutine PROCEDIT is very straightforward due to the division of functions.

After initializing, each scheme on the scratch file is executed. The values in common CONEDIT controls the number and type of editing to be done.

For each scheme the scratch file is read to retrieve the data from the FIELDs clause, if it exists, and the WITH clause, if it exists. The locater information containing the formatted error message and the attribute needed to localize an error and the scheme to be used by routine GETNXT are read in.

GETNXT is then called to retrieve a logical record. For each record XWHERE is called to find out if any editing is to be done. If not, another logical record is read.

The editing begins by executing the data generated by SETFLD from the FIELDs clause if there is any.

For each field the values of HIVAL and LOVAL are compared. If LOVAL is less than HIVAL then the attribute in the common C30 location indicated by LOCAT must be between LOVAL and HIVAL. If HIVAL is less than LOVAL then the attribute must be found in the array LIST between the HIVALth and LOVALth entry. If not, an error message is printed.

If there is a WITH clause function XWITH is used to test each phrase to see if it is valid.

If should be noted that the location message in common LOCATER is printed no more than once for each logical record retrieved by GETNXT.

---

*Main routine of overlay link EPROCE.

At the end of each editing scheme, the number of logical records examined and the number of errors found, is printed.

Figure 77 shows PROCEDIT.

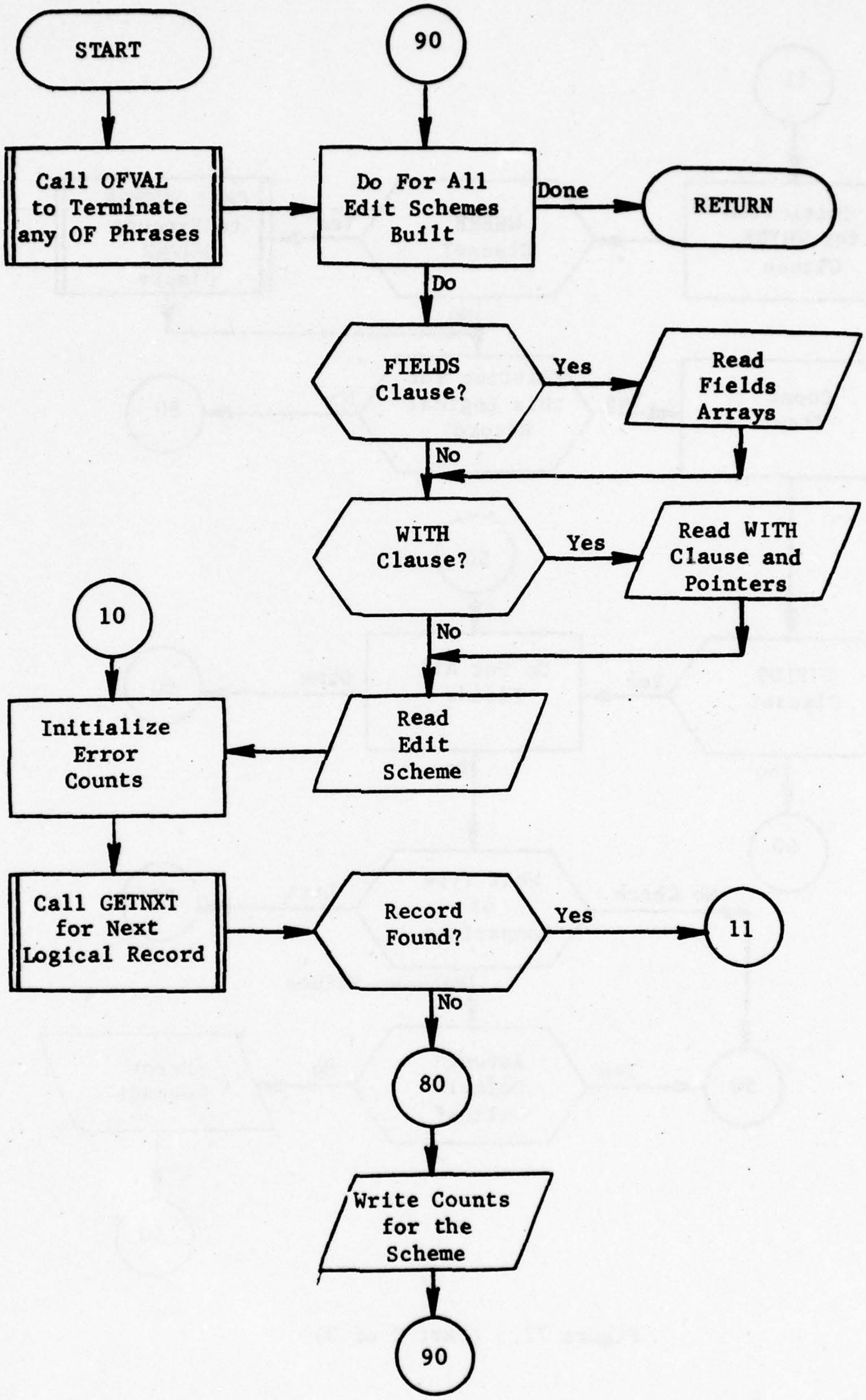



Figure 77. Subroutine PROCEDIT (Part 1 of 3)

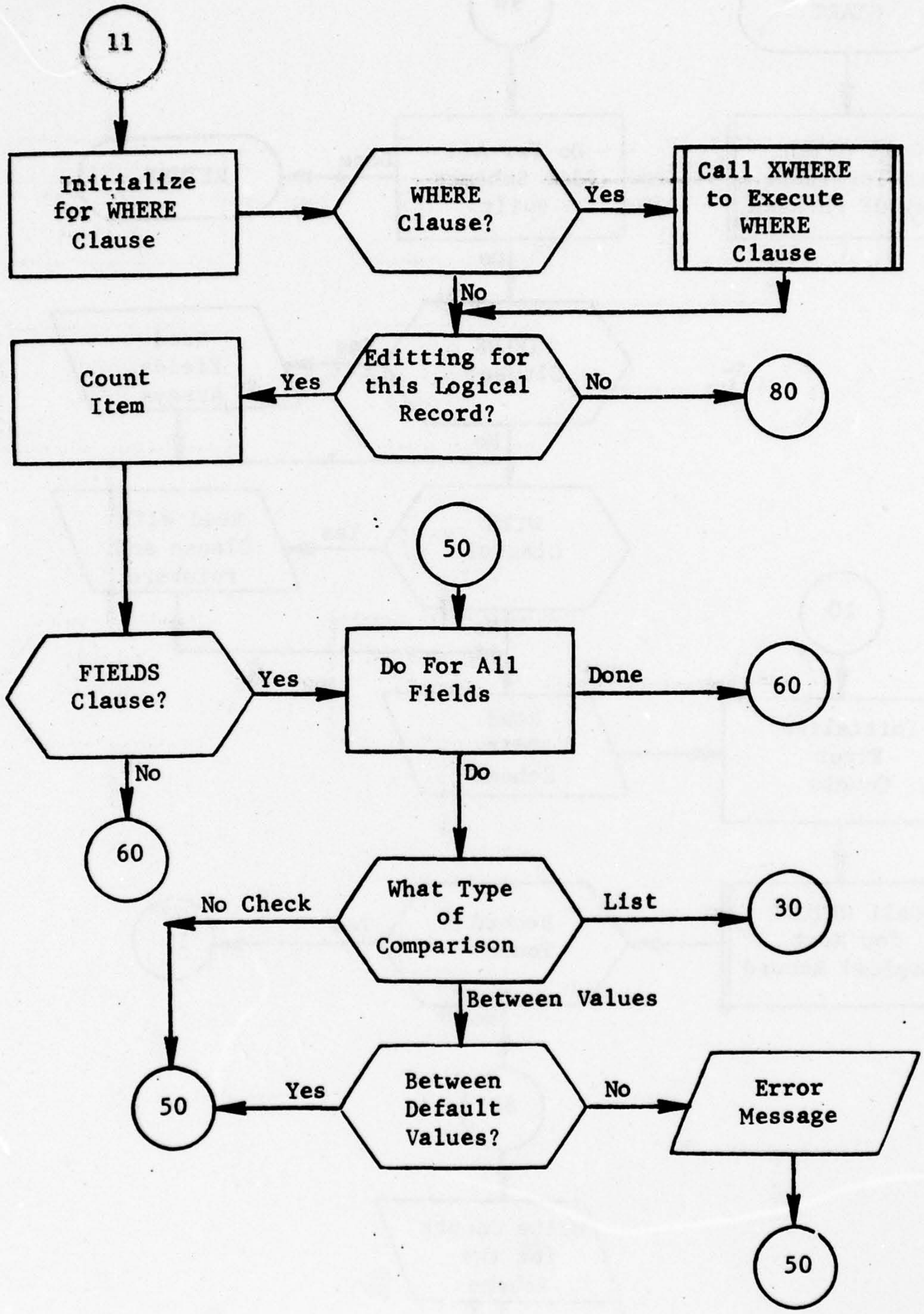



Figure 77. (Part 2 of 3)

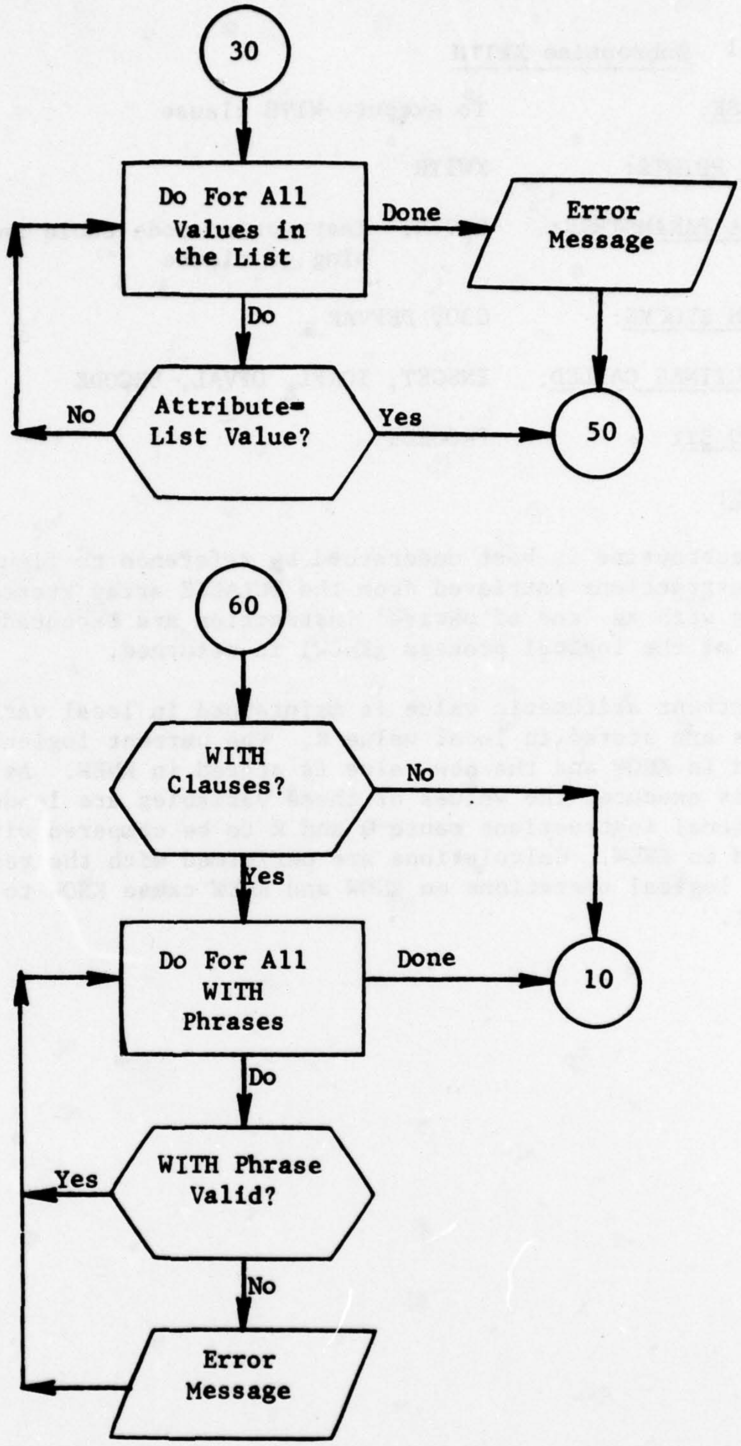



Figure 77. (Part 3 of 3)

5.11.1 **Subroutine XWITH**

PURPOSE: To execute WITH clause

ENTRY POINTS: XWITH

FORMAL PARAMETERS: BEGIN: Instruction code table index for beginning of clause

COMMON BLOCKS: C30, DEFVAR

SUBROUTINES CALLED: INSGET, IORFL, OFVAL, UNCODE

CALLED BY: PROCEDIT

Method:

This subroutine is best understood by reference to figure 78. Basically, the instructions retrieved from the WCLAUSE array starting with BEGIN and ending with an 'end of phrase' instruction are executed. The current value of the logical process (KNOW) is returned.

The current arithmetic value is maintained in local variable Q and new values are stored in local value R. The current logical value is maintained in KNOW and the new value is stored in KNEW. As each instruction is executed the values of these variables are loaded or stored. Relational instructions cause Q and R to be compared with the result placed in KNEW. Calculations are performed with the results placed in Q and logical operations on KNOW and KNEW cause KNOW to have the final result.

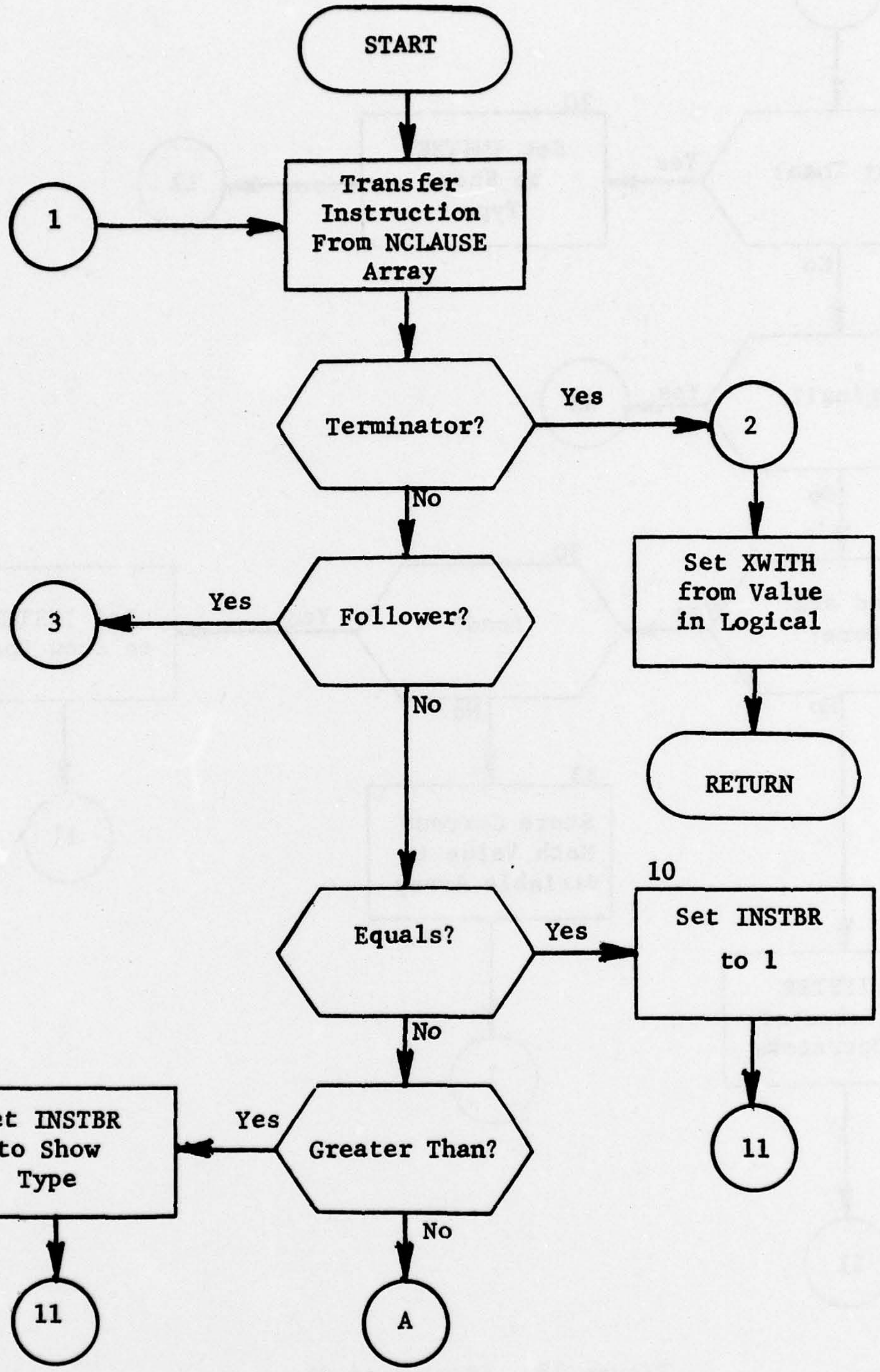



Figure 78. Subroutine XWITH (Part 1 of 9)

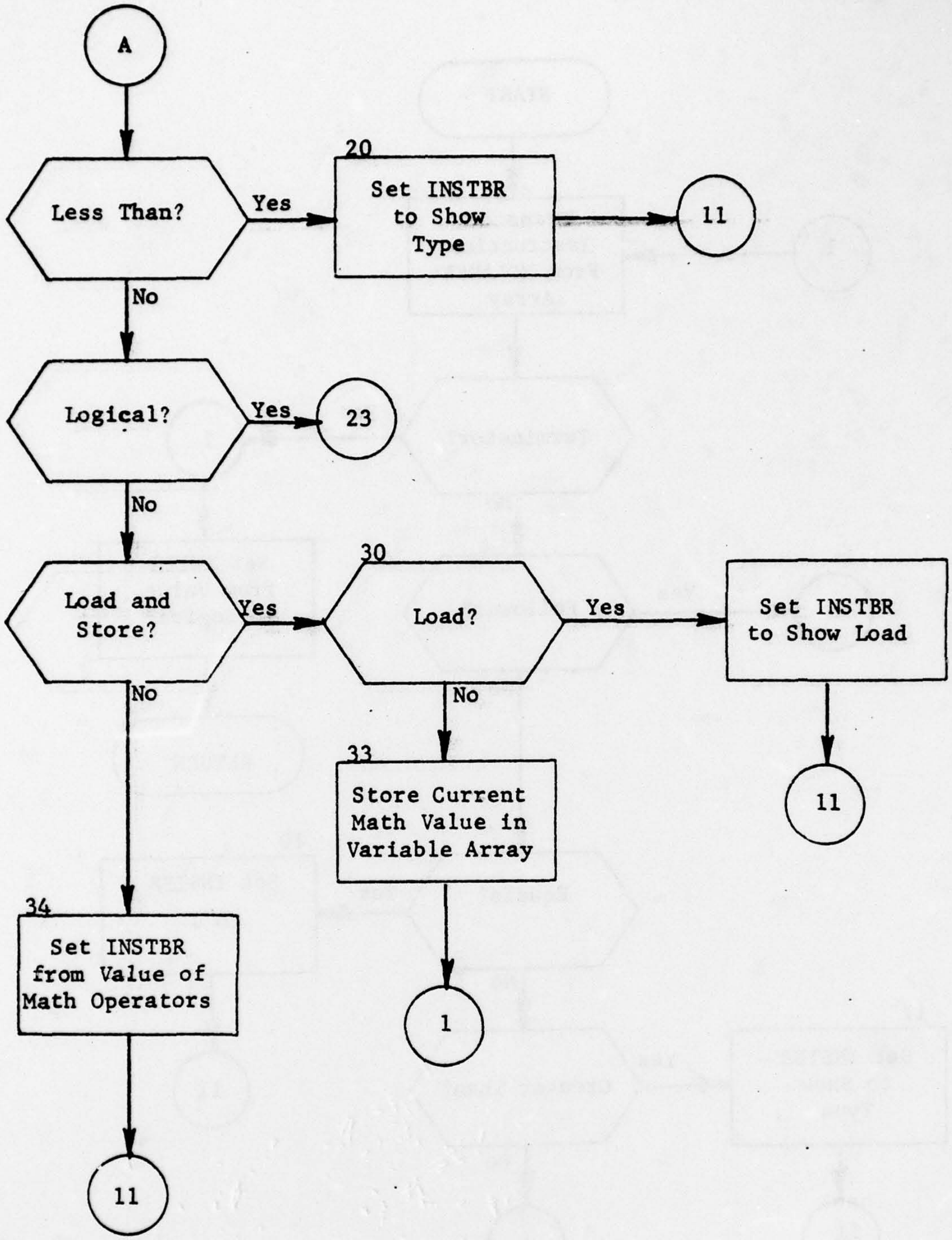



Figure 78. (Part 2 of 9)

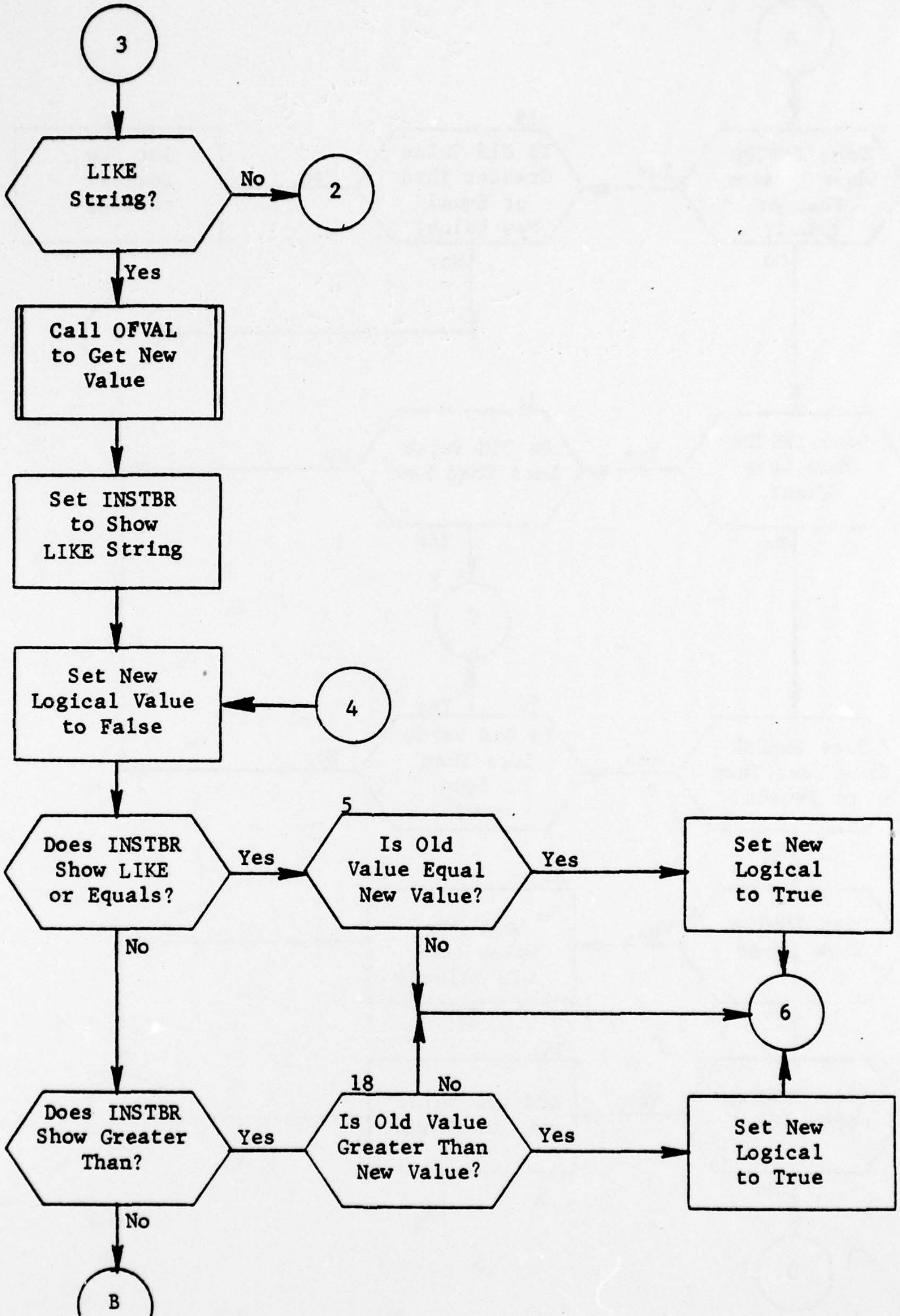



Figure 78. (Part 3 of 9)

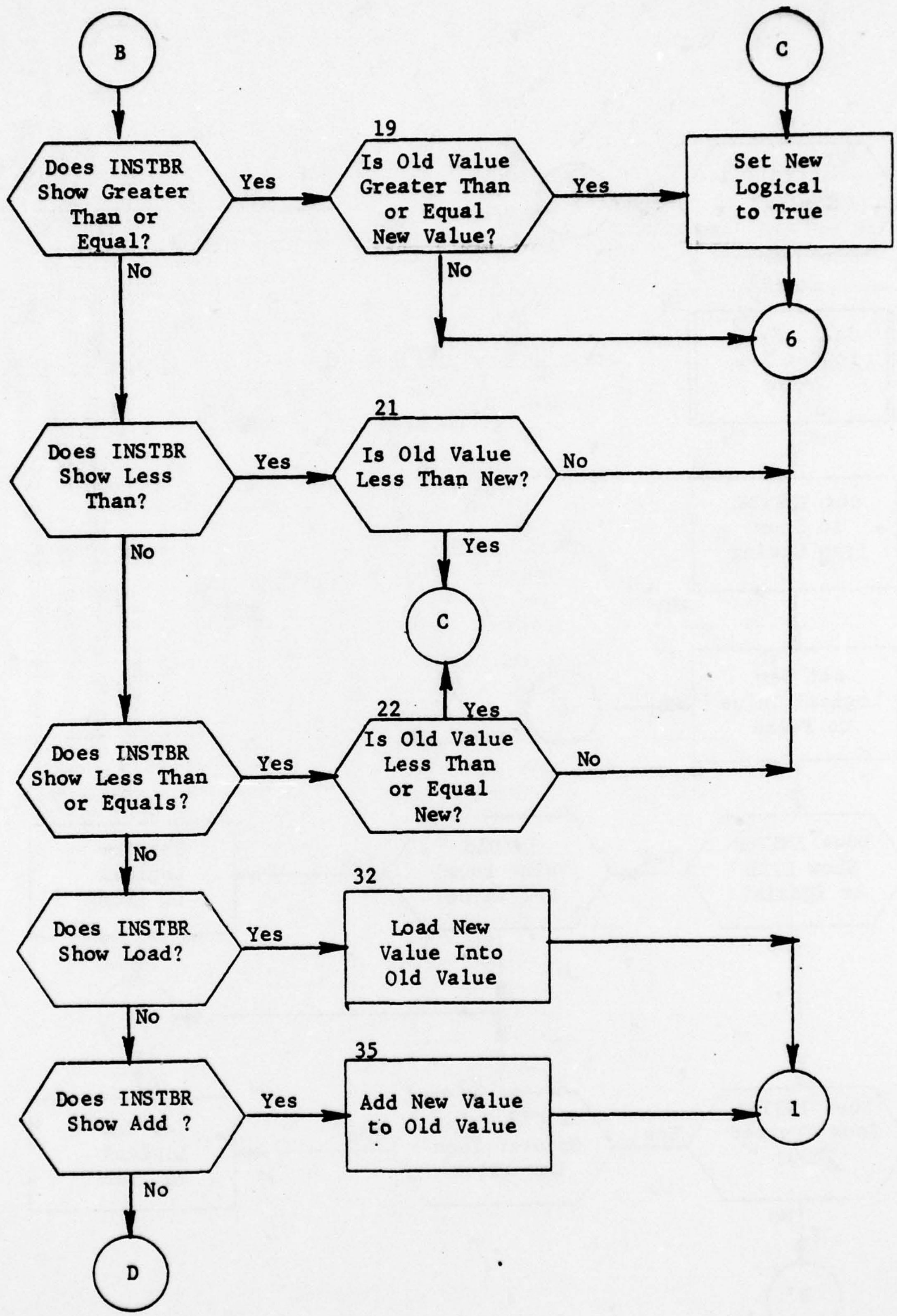



Figure 78. (Part 4 of 9)

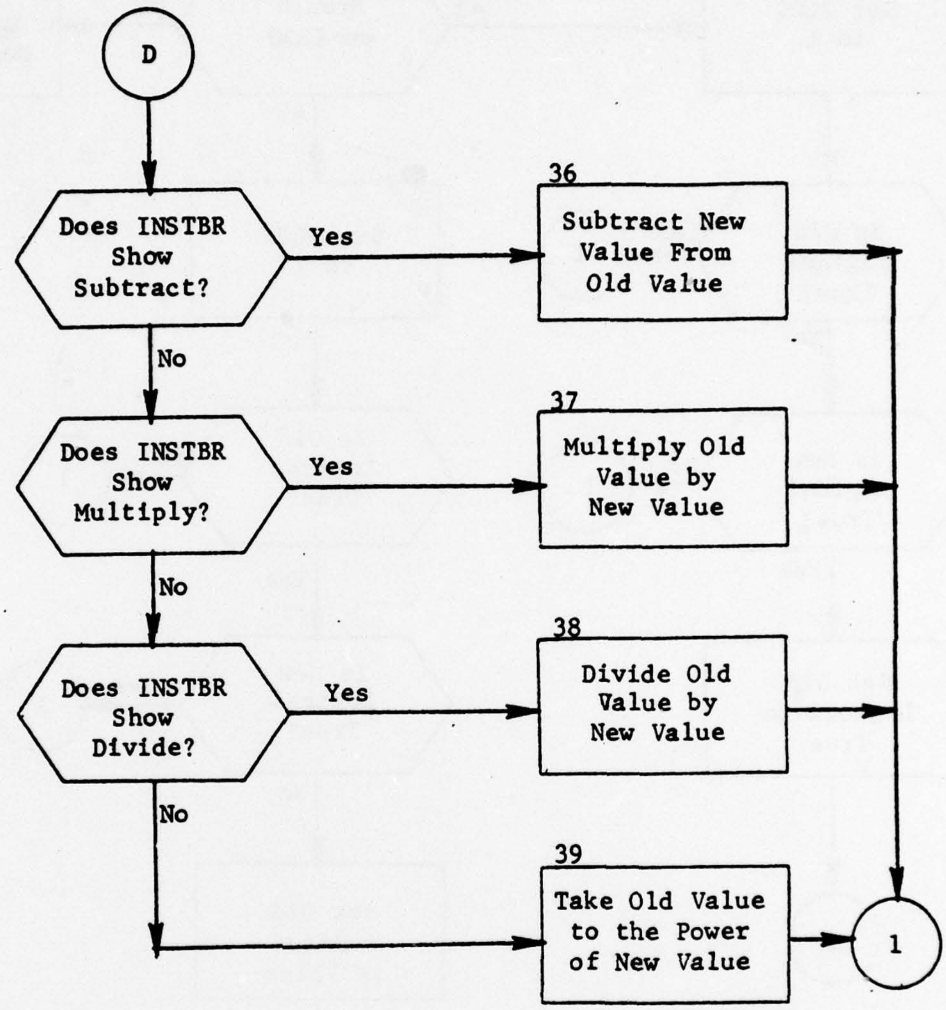



Figure 78. (Part 5 of 9)

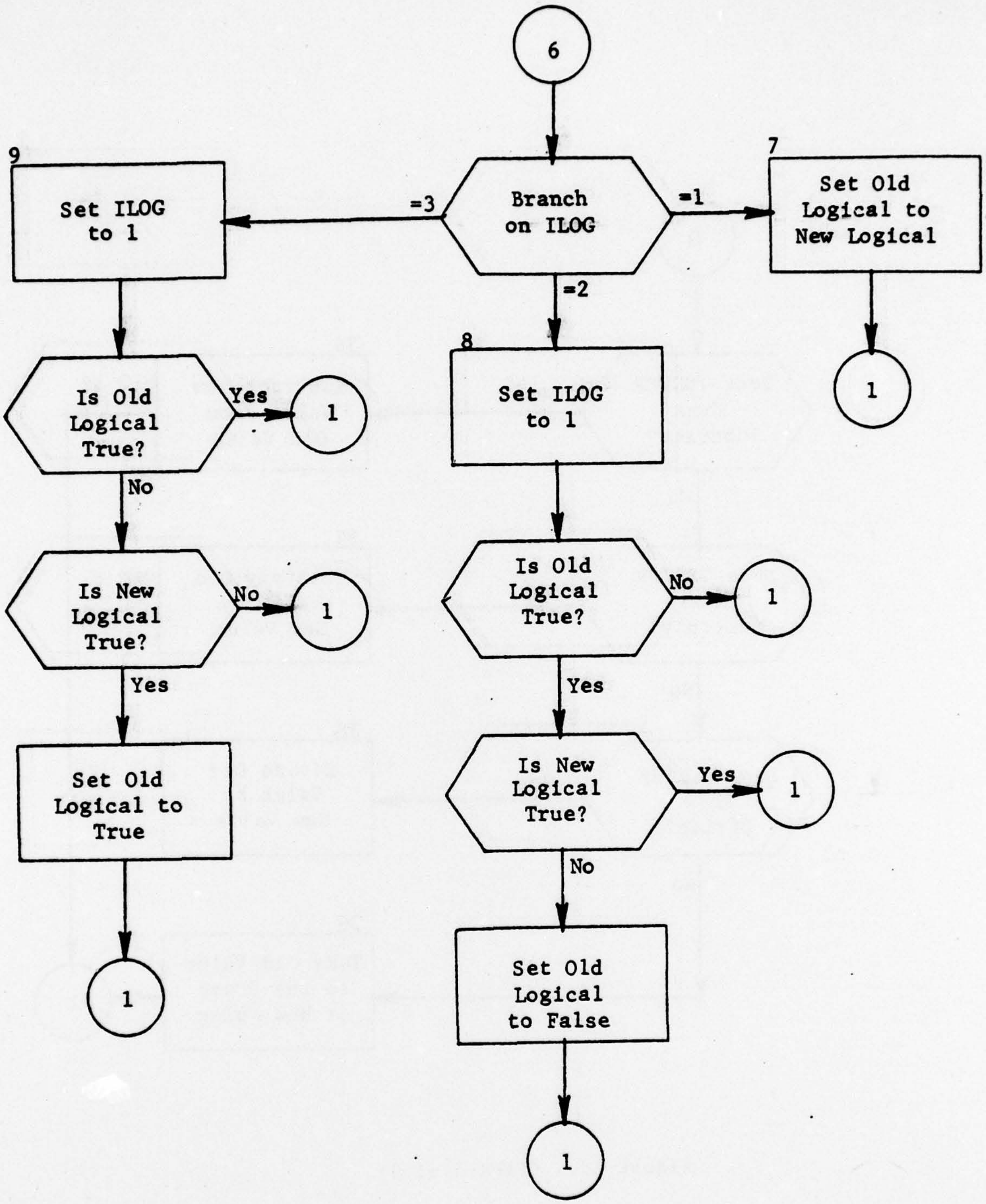



Figure 78. (Part 6 of 9)

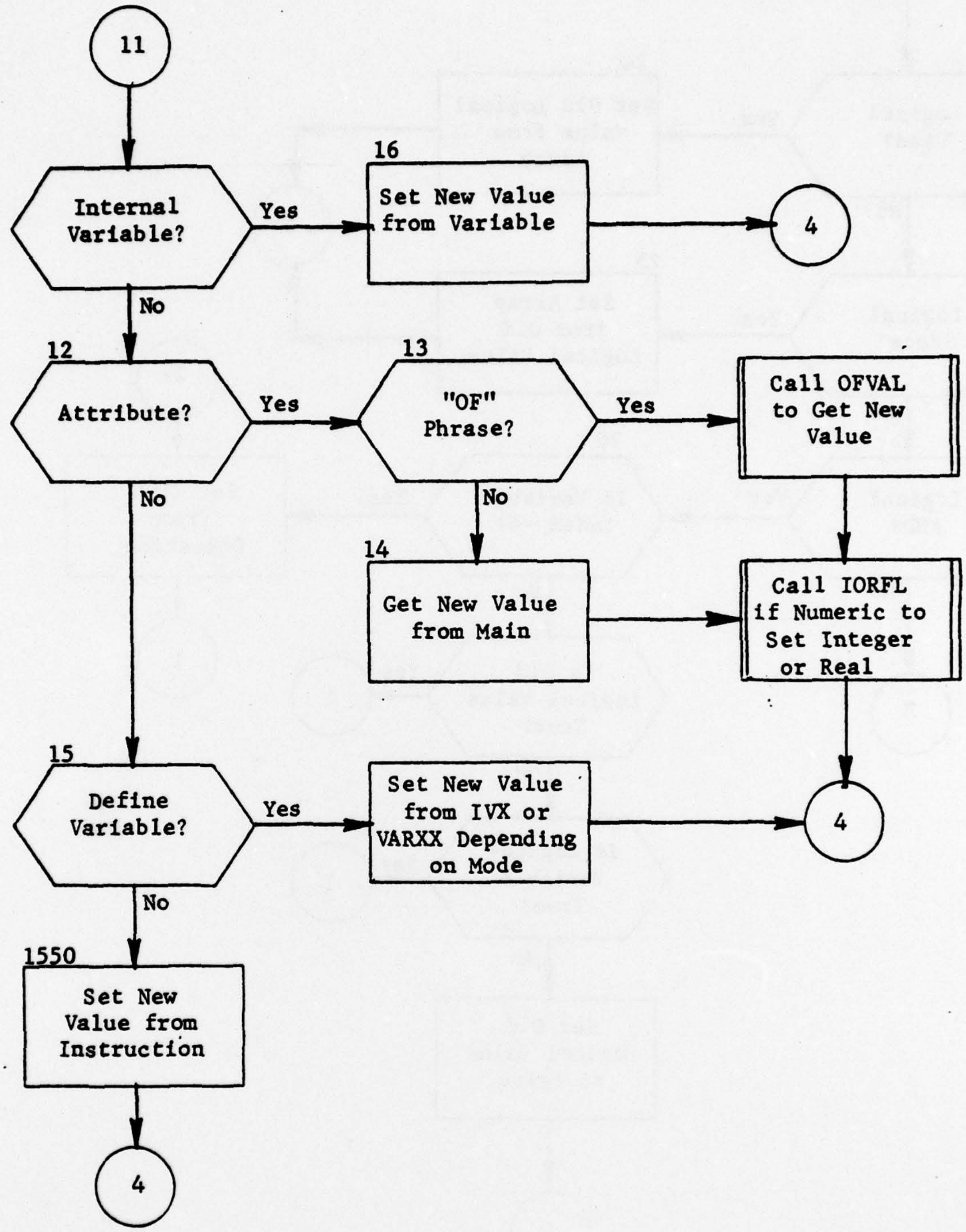



Figure 78. (Part 7 of 9)

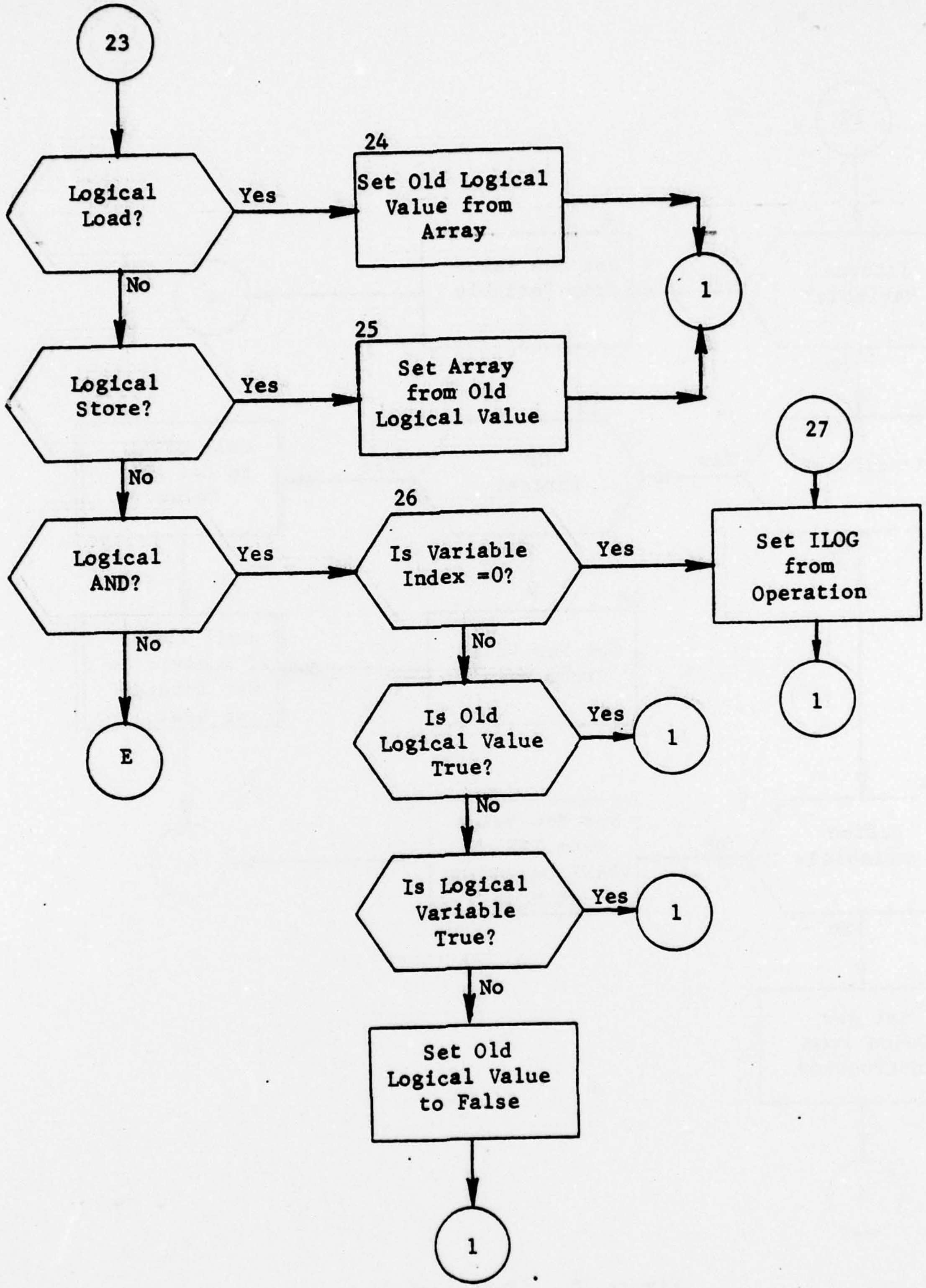



Figure 78. (Part 8 of 9)

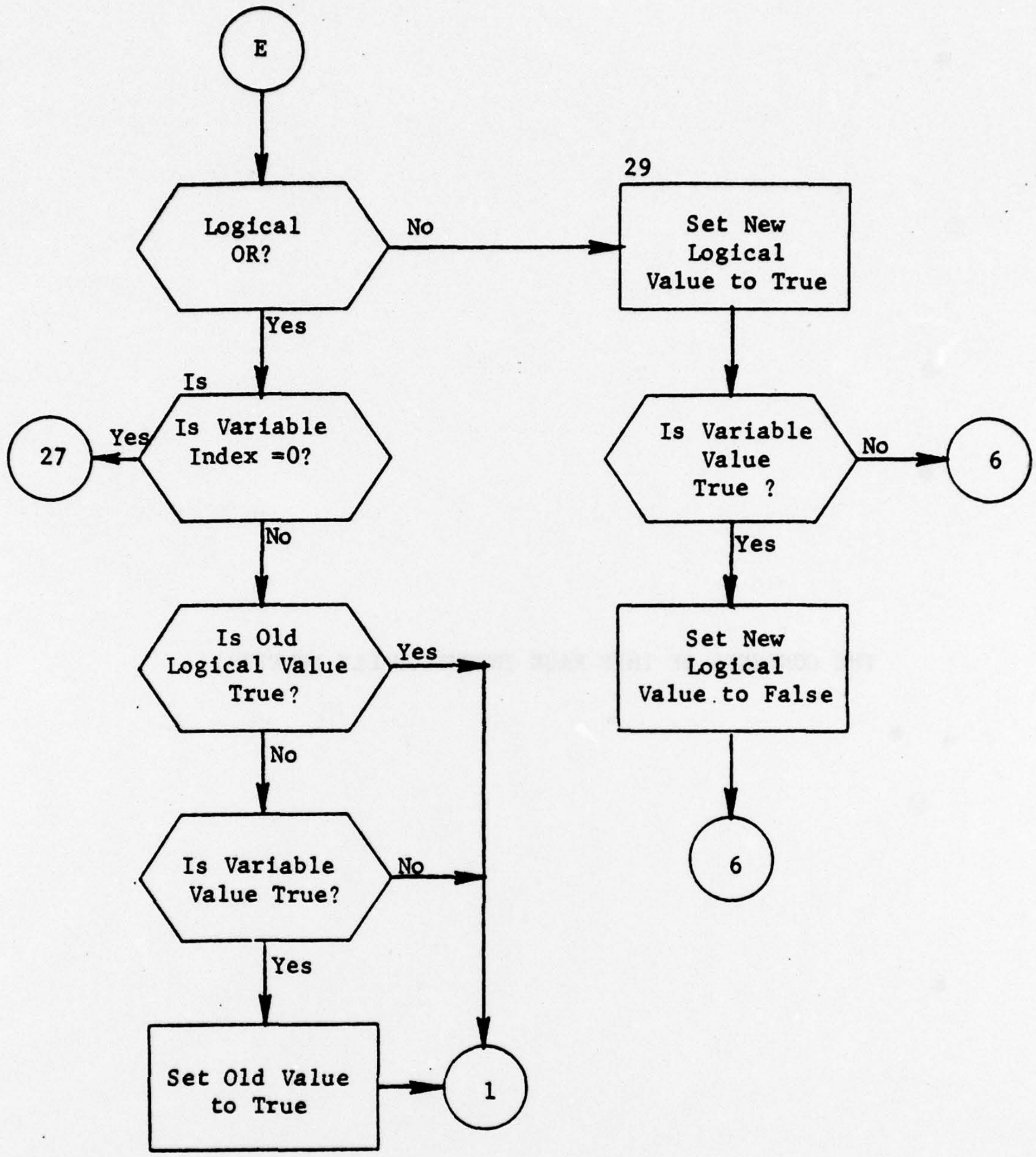



Figure 78. (Part 9 of 9)

Preceding Page Blank - NOT FILMED

# DISTRIBUTION

| Addressee | Copies |
|---|---|
| **CCTC Codes** | |
| Technical Library (C124) | 3 |
| C124 (Stock) | 6 |
| C126 | 1 |
| C313 | 1 |
| C314 | 11 |
| C600 | 1 |
| C730 | 1 |
| **DCA Code** | |
| 205 | 1 |
| **EXTERNAL** | |
| Chief, Studies, Analysis and Gaming Agency, OJCS ATTN: SFD, Room 1D957, Pentagon, Washington, DC 20301 | 5 |
| Chief of Naval Operations, ATTN: OP-96C4, Room 4A478, Pentagon, Washington, DC 20350 | 2 |
| Commander-in-Chief, North American Air Defense Command ATTN: NPXYA, Ent Air Force Base, CO 80912 | 2 |
| Commander, U. S. Air Force Weapons Laboratory (AFSC) ATTN: AFWL/SAB, Kirtland Air Force Base, NM 87117 | 1 |
| Commander, U. S. Air Force Weapons Laboratory (AFSC) ATTN: AFWL/SUL (Technical Library), Kirtland Air Force Base, NM 87117 | 1 |
| Director, Strategic Target Planning, ATTN: (JPS), Offutt Air Force Base, NE 68113 | 2 |
| Defense Documentation Center, Cameron Station, Alexandria, VA 22314 | 12 |
| | 50 |

Preceding Page Blank - Not Filmed

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

# REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

| | |
|---|---|
| 1. REPORT NUMBER: CSM MM 9-77 Volume I, Parts I & II | 2. GOVT ACCESSION NO. |
| 3. RECIPIENT'S CATALOG NUMBER | |
| 4. TITLE (and Subtitle): THE CCTC QUICK-REACTING GENERAL WAR GAMING SYSTEM (QUICK), Program Maintenance Manual, Data Management Subsystem | 5. TYPE OF REPORT & PERIOD COVERED |
| | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s): Dale J. Sanders, Paul F. M. Maykrantz, Jim M. Herrin, Edward F. Bersson | 8. CONTRACT OR GRANT NUMBER(s): DCA 100-75-C-0019 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS: System Sciences, Incorporated, 4720 Montgomery Lane, Bethesda, Maryland 20014 | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS: Command and Control Technical Center, Room BE-685, The Pentagon, Washington, DC 20301 | 12. REPORT DATE: 1 June 1977 |
| | 13. NUMBER OF PAGES: 956 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) | 15. SECURITY CLASS. (of this report): UNCLASSIFIED |
| | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Approved for public release; distribution unlimited.

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

War Gaming, Resource Allocation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The computerized Quick-Reacting General War Gaming System (QUICK) will accept input data, automatically generate global strategic nuclear war plans, provide statistical output summaries, and produce input tapes to simulator subsystems external to QUICK.

The Program Maintenance Manual consists of four volumes which facilitate maintenance of the war gaming system. This volume, Volume I, provides the programmer/analyst with a technical description of the purpose, functions, general procedures, and programming techniques applicable to the modules and subroutines

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED



SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (Cont'd)

of the Data Management Subsystem.

The Program Maintenance Manual complements the other QUICK Computer Manuals to facilitate application of the war gaming system. These manuals (Series 9-77) for Volumes I & II, Series 9-74 for Volumes III & IV) are published by the Command and Control Technical Center (CCTC), Defense Communications Agency (DCA), The Pentagon, Washington, DC 20301.